ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
Τμήμα Ηλεκτρολόγων & Ηλεκτρονικών Μηχανικών
Τμήμα Μηχανικών Βιομηχανικής Σχεδίασης & Παραγωγής
http://www.eee.uniwa.gr
http://www.idpe.uniwa.gr
Θηβών 250, Αθήνα-Αιγάλεω 12241
Τηλ: +30 210 538-1614
Διατμηματικό Πρόγραμμα Μεταπτυχιακών Σπουδών
Τεχνητή Νοημοσύνη και Βαθιά Μάθηση
https://aidl.uniwa.gr/

UNIVERSITY OF WEST ATTICA
FACULTY OF ENGINEERING
Department of Electrical & Electronics Engineering Department of Industrial Design & Production Engineering
http://www.eee.uniwa.gr
http://www.idpe.uniwa.gr
250, Thivon Str., Athens, GR-12241, Greece
Tel: +30 210 538-1614
Master of Science in
Artificial Intelligence and Deep Learning
https://aidl.uniwa.gr/

Master of Science Thesis

Evolutionary Image Generation with Genetic Algorithms and Deep Learning

Student: Despoina Konstantopoulou
Registration Number: AIDL-0042
mscaidl-0042@uniwa.gr

MSc Thesis Supervisor
Paraskevi Zacharia
Assistant Professor

Athens (Egaleo), Greece, October 2024

Supervisor	Member	Member
Paraskevi Zacharia	Michail Papoutsidakis	Helen C. Leligou
Assistant Professor	Professor	Professor/Vice-Director
Department of Industrial Design and Production Engineering	Department of Industrial Design and Production Engineering	Department of Industrial Design and Production Engineering
University of West Attica	University of West Attica	University of West Attica

Copyright © All rights reserved.
University of West Attica and Despoina Konstantopoulou
October, 2024
You may not copy, reproduce or distribute this work (or any part of it) for commercial purposes. Copying/reprinting, storage and distribution for any non-profit educational or research purposes are allowed under the conditions of referring to the original source and of reproducing the present copyright note. Any inquiries relevant to the use of this thesis for profit/commercial purposes must be addressed to the author.
The opinions and the conclusions included in this document express solely the author and do not express the opinion of the MSc thesis supervisor or the examination committee or the formal position of the Department(s) or the University of West Attica.
I, Despoina Konstantopoulou with the following student registration number: AIDL-0042 postgraduate student of the MSc programme in “Artificial Intelligence and Deep Learning”, which is organized by the Department of Electrical and Electronic Engineering and the Department of Industrial Design and Production Engineering of the Faculty of Engineering of the University of West Attica, hereby declare that:
I am the author of this MSc thesis and any help I may have received is clearly mentioned in the thesis. Additionally, all the sources I have used (e.g., to extract data, ideas, words or phrases) are cited with full reference to the corresponding authors, the publishing house or the journal; this also applies to the Internet sources that I have used. I also confirm that I have personally written this thesis and the intellectual property rights belong to myself and to the University of West Attica. This work has not been submitted for any other degree or professional qualification except as specified in it. Any violations of my academic responsibilities, as stated above, constitutes substantial reason for the cancellation of the conferred MSc degree. I wish to deny access to the full text of my MSc thesis until a paper of the same or similar subject is published under Depi Konstantopoulou, Despoina Konstantopoulou or Δέσποινα Κωνσταντοπούλου, following the approval from my supervisor.

The author
Despoina Konstantopoulou

Warm thanks to my mother and her constant support.

In loving memory of my father and my maternal grandparents.

In the context of my thesis, I would like to thank the professor in charge for the preparation of the thesis, Paraskevi Zacharia, *for all her guidance and aid. Furthermore, I would like to thank all the faculty members, past and present, from both the Automation Engineering and the Artificial Intelligence & Deep Learning programs at the University of West Attica.*
In addition, special thanks to my colleagues and friends for their support during the completion of my thesis.

Abstract

This paper introduces GAGAN, a hybrid model that integrates Generative Adversarial Networks (GANs) with Genetic Algorithms (GA) to improve GAN performance in image generation. Traditional GANs often face challenges such as mode collapse and unstable training, where the generator and discriminator struggle to consistently improve. To address these issues, GAGAN employs a hybrid approach: the discriminator’s weights are optimized using GA, while the generator is trained through standard gradient-based backpropagation. The GA evolves the discriminator’s weights, enhancing its ability to distinguish real from fake images, providing more robust feedback to the generator.

This hybrid method combines the exploratory nature of evolutionary algorithms with the efficiency of gradient-based optimization. The model was trained on 2,000 images from the CelebA dataset, generating images at a resolution of 128x128. The results demonstrate that GAGAN outperforms traditional GANs, leading to higher-quality images and more stable convergence. This novel approach enhances adversarial training by leveraging the strengths of both GA and backpropagation techniques.

Keywords

GAGAN, Generative AI, Artificial Intelligence, Machine Learning, Deep Learning, Generative Adversarial Networks (GANs), Genetic Algorithm (GA), Hybrid Model, Discriminator Optimization, Image Generation, Evolutionary Algorithms, Machine Learning Optimization, CelebA Dataset, Mode Collapse, Convergence Stability.

Table of Contents

List of figures……………………………………………………………………………………….10
Acronomy index…………………………………………………………………………….……11

1 Chapter 1: Introduction to Evolutionary Algorithms………………………………….… 12
1.1 Introduction to Computational Intelligence……………………………………………12
1.2 Evolutionary Computation: The Heart of CI…………………………………………..12
1.3 Specialized Algorithms in Evolutionary Computation…………………………………..13
1.4 Mechanisms of Evolutionary Algorithms……………………………………………13
1.5 Adaptability of Evolutionary Algorithms…………………………………………….13
1.6 Balancing Complexity and Simplicity in Evolutionary Algorithms……………………..14
1.7 The Iterative Process of Evolutionary Algorithms…………………………………..14
2 Chapter 2: Basics of Genetic Algorithms (GAs)………………………………………….. 15
2.1 History of Genetic Algorithms……………………………………………………..15
2.2 Genetic Algorithms and Evolutionary Principles……………………………………16
2.3 Guided Random Search and Survival of the Fittest…………………………………16
2.4 Genetic Processes: Selection, Crossover, and Mutation……………………………….16
2.5 Basic Terminology of GAs…………………………………………………………16
2.5.1 Population……………………………………………………………………………...16
2.5.2. Chromosomes…………………………………………………………………………17
2.5.3 Gene…………………………………………………………………………………..17
2.5.4 Allele………………………………………………………………………………….17
2.5.5 Fitness function……………………………………………………………………….17
2.5.6 Genetic operators……………………………………………………………………..17
2.6 How the genetic algorithm works?………………………………………………………...18
2.6.1 Genetic Algorithm Phases: From Initialization to Crossover………………………………18
2.6.2 Genetic Algorithm Phases: From Mutation and Iteration……………………………...19
2.7 Advantages and Limitations of Genetic Algorithms…………………………………...21
2.7.1 Advantages of Genetic Algorithms……………………………………………….21
2.7.2 Limitations of Genetic Algorithms…………………………………………………..22
3 Chapter 3: Introduction to Neural Networks…………………………………………. 23
3.1 History of Neural Networks…………………………………………………………….23
3.1.1. Origins of Neural Networks……………………………………………………….23
3.1.2 Developmental Phases of Neural Networks……………………………………...23
3.1.3 Resurgence in the 1980s……………………………………………………………...23
3.1.4 Renaissance in the 2000s……………………………………………………………..23
3.1.5 Dominance in Modern Applications…………………………………………………24
3.2 What is a Neural Network?………………………………………………………………...24
3.2.1 Definition of Neural Network……………………………………………………………...24
3.2.2 Weights and Biases………………………………………………………………………...24
3.2.2.1 Detailed explanations of the terms related to weights and biases…………………………….25
3.3 Key Neural Network Concepts…………………………………………………………26
3.3.1 Activation Function……………………………………………………………………27

3.3.2 Loss Function……………………28
3.3.3 Regularization……………………28
3.3.4 Parameters (Weights and Biases)……………………28
3.3.5 Bias Values……………………29
3.4 Training Methods for Neural Networks……………………29
3.4.1 Supervised Learning……………………29
3.4.2 Unsupervised Learning……………………29
3.4.3 Reinforcement Learning……………………29
3.5 Types of Neural Networks……………………29
3.5.1 Feedforward Neural Networks (FNN)……………………30
3.5.2 Multilayer Perceptron (MLP)……………………30
3.5.3 Convolutional Neural Networks (CNNs)……………………30
3.5.4 Recurrent Neural Networks (RNNs)……………………30
3.5.6 Long Short-Term Memory (LSTM)……………………30
3.6 Forward Propagation and backpropagation in NNs……………………31
3.6.1 Forward propagation……………………31
3.6.2 Back propagation……………………31
3.6.3 Comparing Forward and backpropagation……………………32
4 Chapter 4: Introduction to Convolutional Neural Networks (CNNs)……………………33
4.1 Definition of a CNN……………………33
4.2 Core Elements of a Convolutional Neural Network……………………34
4.2.1 Convolutional Layers……………………34
4.2.2 Rectified Linear Unit (ReLU)……………………34
4.2.3 Pooling Layers……………………34
4.2.4 Fully Connected Layers……………………36
4.2.5 Python Code for a CNN Model……………………36
4.3 Underfitting and Overfitting in convolutional neural network (CNNs)……………………37
4.3.1 Underfitting in CNNs……………………37
4.3.1.1 Exploring Underfitting in CNNs……………………37
4.3.1.2 Strategies to Overcome Underfitting……………………38
4.3.2 Overfitting in CNNs……………………38
4.3.2.1 Causes of Overfitting in CNNs……………………38
4.3.2.2 Strategies for Mitigating Overfitting……………………38
5 Chapter 5: Overview of Generative Adversarial Networks (GANs)……………………40
5.1 Introduction to GANs……………………40
5.2 The main components of a GAN……………………40
5.2.1 The Role of the Generator in GANs……………………41
5.2.1.1 A basic generator model example in Python code……………………41
5.2.2 The Role of the Discriminator in GANs……………………42
5.2.2.1 A basic generator model example in Python code……………………42
5.2 How Do GANs work?……………………42
5.2.1 Training the generator and discriminator……………………43
5.2.2 Mathematical Explanation of Training……………………43
5.3 Applications of GANs……………………44

5.4 Types of GANs....45
6 Chapter 6: Deep Convolutional GANs (DCGANs)....46
6.1 Introduction to DCGANs....46
6.1.1 Architectural Innovations....46
6.1.2 Image Generation Process....46
6.1.3 Challenges and Solutions....46
6.1.4 Practical Application....46
6.1.5 Algorithm for Minibatch Stochastic Gradient Descent in GANs....47
6.2 Key differences between GANs and DCGANs....48
6.2.1 Architecture....48
6.2.2 Image Generation....49
6.2.3 Input Handling....49
6.2.4 Pooling Techniques....49
6.2.5 Batch Normalization....49
6.2.6 Activation Functions....49
6.2.7 Training Stability....50
6.2.8 Spatial Hierarchies....50
6.2.9 Image Resolution....50
7 Chapter 7: The GAGAN - Combining GAs with GANs.... 51
7.1 Introduction to GAGAN Hybrid....51
7.1.1 Leveraging Genetic Algorithms in GANs....51
7.1.2 Advantages of the GAGAN Synergy....51
7.1.3 Exploring the GAGAN Mechanisms.... 51
7.2 Mechanism of GAGAN Hybrid....51
7.2.1 Individual Representation....51
7.2.2 Fitness Function Definition....52
7.2.2.1 Performance Metrics in GAGAN....52
7.2.2.2 Training Evaluation and Selection Process in GAGAN....53
7.3 Genetic Operators in GAGAN....53
7.3.1 Selection Process....53
7.3.2 Crossover Techniques....53
7.3.3 Mutation Strategies....54
8 Chapter 8: GAGAN in Action: Experimental Insights....55
8.1 Introduction....55
8.2 Initial Challenges....55
8.3 Experiment Setup....55
8.4 Approach to Optimization....55
8.5 Final Parameter Configuration....56
8.6 Integrated Numerical and Visual Analysis of GAGAN and DCGAN Models....56
8.6.1 Numerical Analysis....56
8.6.1.1 Final Losses....56
8.6.1.2 Observations....57
8.6.2 Visual Comparison....58
8.6.2.1 Images Generated with GAGAN....58

8.6.2.2 Images Generated with DCGAN……60
8.6.2.3 Visual Outputs……60
8.7 Results and Observations……61
8.8 Conclusion……61
9 Chapter 9: Challenges and Future Directions…… 62
9.1 Challenges……62
9.1.1 Computational Constraints and Resource Limitations……62
9.1.2 Balancing Dataset Size and Training Time……62
9.1.3 Parameter Selection and Model Tuning……62
9.2 Future Directions……63
9.2.1 Advanced Genetic Algorithms……63
9.2.2 Integrating Perceptual Metrics for Fitness Evaluation……63
9.2.3 Expanding Genetic Optimization to the Generator……63
Conclusions…… 64
Bibliography- References- Papers- Online Sources……65

List of Figures

Figure 1: Computational Intelligence categories with different methods……[12]
Figure 2: Partial Taxonomy of Common Evolutionary Computation Methods……[13]
Figure 3: Process of Evolutionary Algorithms……[14]
Figure 4: Schema of Genetic Algorithms……[15]
Figure 5: Population, chromosome, and genes in the genetic algorithm……[17]
Figure 6: Pseudocode of classical genetic algorithm……[18]
Figure 7: Crossover operation……[19]
Figure 8: Mutation example……[20]
Figure 9: Genetic Algorithm Phases……[20]
Figure 10: Operators in GAs……[21]
Figure 11: A simple perceptron……[24]
Figure 12: Structure of Typical Neuron and Artificial Neuron……[25]
Figure 13: Basic structure of the layers of an Artificial Neural Network (ANN)……[26]
Figure 14: Activation Functions……[27]
Figure 15: Forward propagation……[31]
Figure 16: Backpropagation……[32]
Figure 17: Basic structure of a CNN……[33]
Figure 18: Pooling method for convolutional neural networks……[35]
Figure 19: Max Pooling example……[35]
Figure 20: Example of a CNN sequence to classify handwritten digits……[36]
Figure 21: Overfitting & Underfitting example……[37]
Figure 22: Basic structure of GAN……[40]
Figure 23: General workflow of GAN architecture……[41]
Figure 24: Block diagram of GAN……[43]
Figure 25: Mathematical Explanation of Training……[44]
Figure 26: DCGAN architecture……[47]

Figure 27: Minibatch stochastic gradient descent training of GANs.……………….…………………..[48]
Figure 28: Training Process of GAGAN……………..……………..……………..…………………...[56]
Figure 29: Generated Images with GAGAN…….……………..……………..……………..…………[59]
Figure 30: Generated Images with DCGAN….……………..……………..……………..…………….[60]

Acronym Index:

AIDL - Artificial Intelligence & Deep Learning
AI - Artificial Intelligence
AIS - Artificial Immune Systems
ACO - Ant Colony Optimization
ANN - Artificial Neural Network
CI - Computational Intelligence
DCGAN - Deep Convolutional Generative Adversarial Network
DL - Deep Learning
EAs - Evolutionary Algorithms
EC - Evolutionary Computation
EP - Evolutionary Programming
ES - Evolutionary Strategies
GA - Genetic Algorithm
GAGAN - Genetic Algorithm-GAN Hybrid
GP - Genetic Programming
GPU - Graphics Processing Unit
LSTM - Long Short-Term Memory
Leaky ReLU - Leaky Rectified Linear Unit
ML - Machine Learning
MSE - Mean Squared Error
MLP - Multilayer Perceptron
NN - Neural Network
NGEN - Number of Generations
PSO - Particle Swarm Optimization
RAM - Random Access Memory
ReLU - Rectified Linear Unit
RNN - Recurrent Neural Network
SVM - Support Vector Machine
TPU - Tensor Processing Unit
cGAN - Conditional Generative Adversarial Network
WGAN - Wasserstein GAN

1. Introduction to Evolutionary Algorithms

1.1 Introduction to Computational Intelligence

In an era where data-driven technologies are becoming increasingly vital to numerous industries, the concept of computational intelligence (CI) emerges as a pivotal framework. Computer intelligence learning (CI) encompasses a computer's ability to learn and adapt through data or experimental observation, thus enabling machines to perform complex tasks. Despite being used synonymously with soft computing, CI still lacks a consensus definition that takes into account its wide range of applications and approaches. [1], [2]

1.2 Evolutionary Computation: The Heart of CI

At the heart of CI lies evolutionary computation (EC), a dynamic subset that draws inspiration from biological, ecological, and cultural processes. This framework includes various approaches, such as evolutionary algorithms (EAs), which are optimization techniques based on the principles of natural selection; neural networks (NNs), which are computational models inspired by the human brain's structure and function and are particularly effective in pattern recognition; fuzzy logic, a form of many-valued logic that deals with reasoning that is approximate rather than fixed and exact; swarm intelligence, a collective behavior exhibited by decentralized systems, such as particle swarm optimization (PSO), which simulates social behavior in animals; and artificial immune systems (AIS), which are inspired by the biological immune system's processes for detecting and responding to pathogens. Each of these methodologies offers unique mechanisms for addressing optimization, learning, and problem-solving tasks, allowing for a broad range of applications.

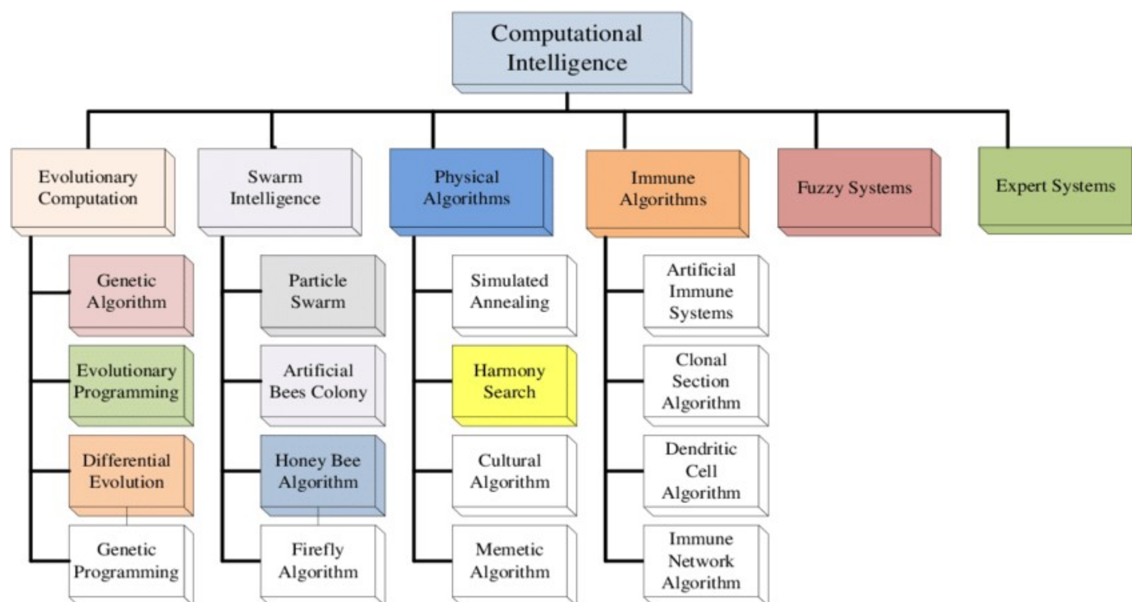


Figure 1: Computational Intelligence categories with different methods, [2].

1.3 Specialized Algorithms in Evolutionary Computation

Evolutionary computation itself comprises several specialized algorithms, each designed to tackle specific challenges. Key categories include genetic algorithms (GAs), which mimic the process of natural selection to evolve solutions; genetic programming (GP), which evolves computer programs to solve problems; evolutionary strategies (ES), which focus on optimizing the parameters of real-valued functions; and evolutionary programming (EP), which is designed for the optimization of control parameters in machine learning. Additionally, swarm intelligence techniques like PSO and ant colony optimization (ACO), which simulates the foraging behavior of ants to find optimal paths, further illustrate the diversity within this domain. Each subset leverages distinct principles from nature, enabling the exploration of complex solution spaces. [3], [4]

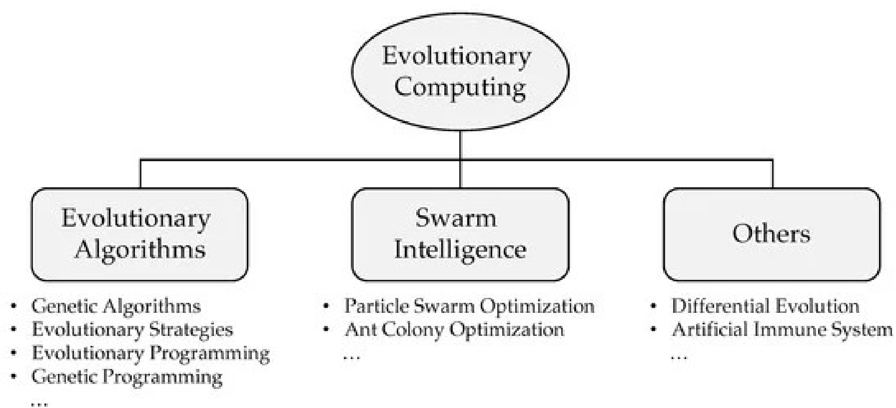


Figure 2: Partial Taxonomy of Common Evolutionary Computation Methods, [3]

1.4 Mechanisms of Evolutionary Algorithms

Central to the effectiveness of evolutionary algorithms is their ability to mimic natural evolution processes such as reproduction, mutation, recombination, and selection. These processes facilitate the development of solutions over time, where a population of potential candidates—referred to as individuals—is assessed against a fitness function. This function evaluates each candidate's suitability for a given task, akin to a loss function in machine learning. By iteratively applying evolutionary operators, the population evolves through cycles that aim to enhance the quality of solutions in each generation. [5]

1.5 Adaptability of Evolutionary Algorithms

A significant advantage of EAs is their remarkable adaptability. They do not rely on predefined notions

about problem structures, making them well-suited for addressing complex, multimodal, and poorly understood optimization problems. Through genetic variety and population diversity, these algorithms can navigate vast solution spaces, effectively avoiding premature convergence on suboptimal solutions.

1.6 Balancing Complexity and Simplicity in Evolutionary Algorithms

Although evolutionary algorithms (EAs) are effective methods for resolving optimization issues, they have certain drawbacks, most notably with regard to computing complexity. It can require a lot of resources to repeatedly evaluate fitness functions, particularly when dealing with big populations in real-world situations. In order to overcome this, researchers frequently use fitness approximation techniques like heuristic shortcuts and surrogate modeling, which lessen the requirement for precise fitness evaluations and increase computational efficiency. It's interesting to note that an EA's ability to address complicated problems is not always reflected in how simple it is. It has been demonstrated that even simple EAs are able to handle complex optimization tasks, demonstrating their adaptability and resilience. [5], [6]

1.7 The Iterative Process of Evolutionary Algorithms

In essence, evolutionary algorithms operate by generating a population of candidate solutions, assessing their fitness, and iteratively refining them through selection, recombination, and mutation. This process, inspired by the principles of natural selection and genetic variation, enables EAs to evolve towards increasingly effective solutions over successive generations. As each iteration unfolds, the algorithm maintains a population of solutions, evaluates their fitness, selects the fittest individuals, and ultimately creates a new population for the next cycle. [5], [7]

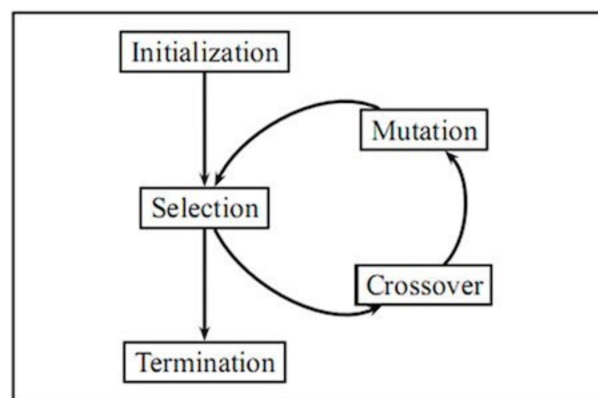


Figure 3: Process of Evolutionary Algorithms , [7]

2. Basics of Genetic Algorithms (GAs)

In computer science, a simple genetic algorithm is an exploratory search and optimization process that uses genetic operations such as crossover, mutation, and reproduction on a population of genotype strings to find answers to problems. This process resembles natural evolution. [8]

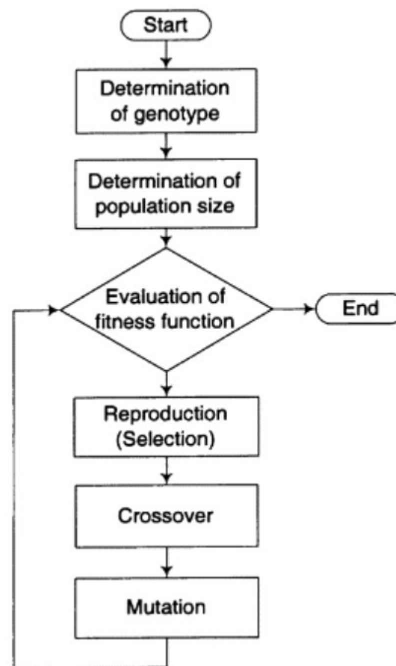


Figure 4: Schema of Genetic Algorithms, [8]

2.1 History of Genetic Algorithms

In 1950, Alan Turing introduced the concept of a "learning machine" which would parallel the principles of evolution. [9] In 1954, Nils Aall Barricelli conducted early computer simulations of evolution, laying the groundwork for future research. In 1957, Australian geneticist Alex Fraser published papers on artificial selection with multiple loci, incorporating key elements of modern GAs. The 1960s saw increased interest, with contributions from researchers like Hans-Joachim Bremermann, who focused on population-based solutions for optimization. Ingo Rechenberg and Hans-Paul Schwefel's work on evolutionary techniques for complicated engineering issues led to the widespread recognition of GAs. Interest in GAs was greatly increased by John Holland's 1975 book, Adaptation in Natural and Artificial Systems, which presented a paradigm for predicting generational quality. GAs were originally commercialized in the late 1980s, starting with General Electric's industrial process toolkit and continuing with Axcelis, Inc.'s 1989 release of Evolver, the first GA product for desktop computers that attracted media attention. Since then, GAs and other heuristic optimization techniques have been integrated into platforms such as MATLAB, expanding their use and accessibility in a variety of domains. [9], [10]

2.2 Genetic Algorithms and Evolutionary Principles

Genetic Algorithms (GAs) are a class of adaptive heuristic search algorithms rooted in the principles of genetics and natural selection, designed to solve optimization and search problems. They are based on the concepts of genetics and natural selection. The fundamental principle of GAs is to identify the best or almost optimal solution for a given issue by gradually evolving a population of candidate solutions, sometimes called individuals or chromosomes, over several generations. GAs are very good at addressing complicated problems because they use evolutionary mechanisms to gradually improve solutions. [10], [11]

2.3 Guided Random Search and Survival of the Fittest

GAs intelligently explore the solution space by combining historical data to direct the search toward higher-performing regions, in contrast to merely random searches. Motivated by the idea of natural selection, individuals possessing superior qualities—traits better suited to their surroundings—are more likely to survive, procreate, and transfer those traits to the following generation. The way genetic algorithms (GAs) work is reminiscent of this biological metaphor: they mimic the "survival of the fittest" by gradually improving and optimizing potential solutions through successive generations.

2.4 Genetic Processes: Selection, Crossover, and Mutation

Each individual in a GA population, much like a biological chromosome, represents a possible solution encoded as a string of characters, numbers, floats, or bits. Through genetic processes such as selection, crossover (recombination), and mutation, the population evolves, continually improving its overall fitness. This evolutionary cycle produces high-quality solutions, making GAs a powerful tool for tackling complex optimization and search challenges.

2.5 Basic Terminology of GAs

In order to fully understand how Genetic Algorithms (GAs) operate, it’s essential to grasp some basic terminologies that define the process and elements involved in this algorithm.

2.5.1 Population:

A group of potential solutions at any particular algorithmic iteration is referred to as the population. It is a subset of every potential or likely solution to the issue at hand. Each person within the population represents a different potential solution, and the algorithm aims to improve this population over future generations by performing various genetic operations.

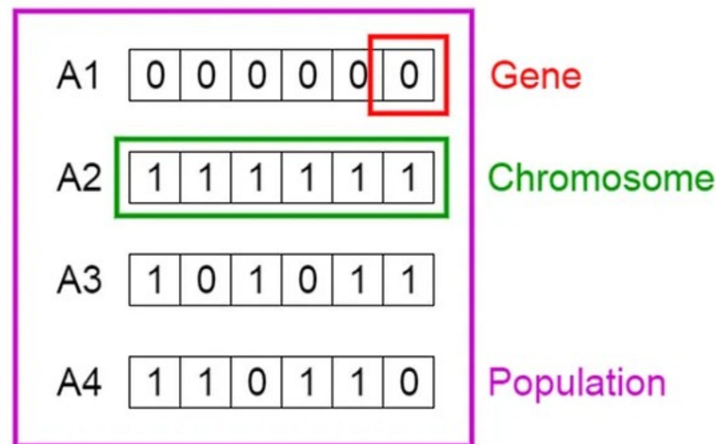


Figure 5: Population, chromosome, and genes in the genetic algorithm, [14]

2.5.2. Chromosomes:
Within the population, a chromosome represents a single potential solution. It can be compared to a blueprint that contains an encoded specific solution to the issue at hand. A chromosome in a GA is usually represented as a string of variables or symbols (real numbers, integers, or binary digits). Each of these variables contributes to the total solution, and they are placed in a particular order.

2.5.3 Gene:
A gene is a single chromosomal piece or component. In the encoded solution, it stands for a certain characteristic or variable. Each gene on the entire chromosome is in charge of regulating a distinct component of the solution. Genes are the fundamental components that go into creating the whole solution in a genetic environment.

2.5.4 Allele:
The unique value that a gene on a chromosome is assigned is called an allele. Similar to how an allele in biology designates a particular gene variation, an allele in GAs designates the particular value a gene can have. An allele could indicate if a property, like "length," is determined by a gene, for instance, and whether it is long or short.

2.5.5 Fitness function:
An important factor in determining how "fit" or competent each member of the population is at addressing the problem is the fitness function. Every chromosome is given a fitness score according to how well it performs in relation to the target. The fittest individuals—those with the highest fitness scores—are more likely to be chosen for reproduction and passed on to the following generation. Individuals are evaluated using this fitness function in each algorithm iteration.

2.5.6 Genetic operators:
The mechanisms that modify the genetic composition of the following generation are known as genetic operators. Among these are mutation, crossover, and selection.

- Selection makes decisions about who in the population is most qualified to raise the next generation of people.
- To create kids, crossover (recombination) joins the genetic material of two parent chromosomes.
- Mutation creates haphazard alterations into progeny's genes in order to preserve variety and investigate novel therapeutic options.

  In order for the algorithm to produce children that may surpass the parent solutions, these operators are essential to the GA's capacity to explore and utilize the solution space.

2.6 How the genetic algorithm works?

The genetic algorithm operates in five main stages, followed by a cycle of repetition that continues until specific conditions are either met or unmet, depending on the requirements.

Input:
Population Size, n
Maximum number of iterations, MAX
Output:
Global best solution, Y_{bt}

begin
Generate initial population of n *chromosomes* Y_i $(i = 1, 2, ..., n)$
Set iteration counter $t = 0$
Compute the fitness value of each chromosomes
while $(t < MAX)$
Select a pair of chromosomes from initial population based on fitness
Apply crossover operation on selected pair with crossover probability
Apply mutation on the offspring with mutation probability
Replace old population with newly generated population
Increment the current iteration t *by 1.*
end while
return the best solution, Y_{bt}
end

Figure 6: Pseudocode of classical genetic algorithm, [13]

2.6.1 Genetic Algorithm Phases: From Initialization to Crossover

The genetic algorithm begins with initialization, where a population of random solutions, known as chromosomes, is generated. These chromosomes are typically represented as binary strings, each encoding a potential solution to the given problem. Once the population is initialized, the next step is evaluation, where each chromosome is assessed for its fitness. Fitness refers to how well the chromosome solves the problem, with higher fitness indicating a better solution.

Following evaluation, selection takes place. In this phase, a subset of chromosomes is chosen based on their fitness scores, using methods such as fitness-proportionate selection. Chromosomes with higher fitness are more likely to be selected for the next phase, though less fit chromosomes might still be chosen, maintaining diversity in the population.

One of the most critical phases of a genetic algorithm is crossover, where selected chromosomes are paired and combined to produce offspring. This process involves exchanging sections of genetic material between the parents at a random crossover point. For instance, if a crossover point is chosen at position 3, the parents' genetic material up to that point is swapped, generating new offspring. There are different types of crossover methods. In single-point crossover, all genes are exchanged after a randomly chosen crossover point. Multi-point crossover involves selecting multiple points where genes are transferred between the two parents, while uniform crossover randomly selects genes from either parent with an associated probability, allowing for greater variation in the offspring. These newly generated offspring are added to the population, moving the algorithm toward the next iteration of evolution.

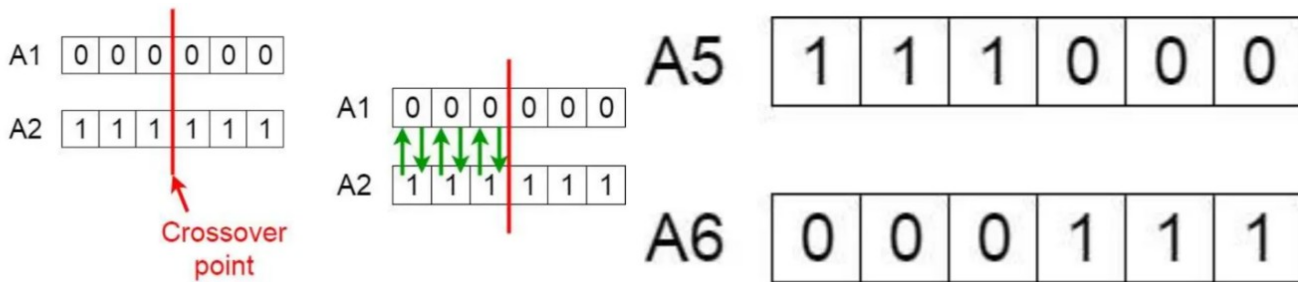


Figure 7: Crossover operation, [15]

2.6.2 Genetic Algorithm Phases: From Mutation and Iteration

The mutation phase is crucial for maintaining genetic diversity within the population. Mutation introduces random modifications to the genes of offspring, preventing the population from becoming too similar and avoiding premature convergence on suboptimal solutions. This is typically done through techniques such as chromosome flipping, where specific bits in a bit string are flipped at random. The probability of mutation is kept low, ensuring that mutations occur sporadically but effectively. Various mutation techniques include flip bit mutation, where random bits in the chromosome are flipped; Gaussian mutation, which adds a small, randomly generated Gaussian-distributed value to certain genes; and exchange/swap mutation, where the positions of two genes within the chromosome are swapped. These mutation methods ensure the algorithm explores new

solutions while maintaining diversity in the population, thus enhancing the search for optimal outcomes.

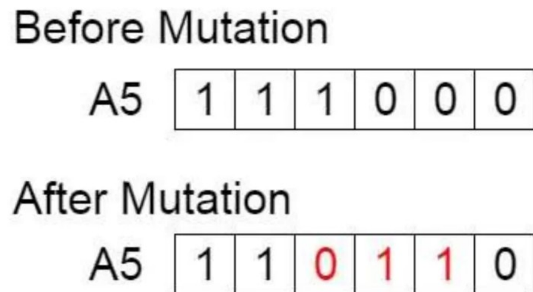


Figure 8: Mutation example, [15]

The genetic algorithm then enters a repeat phase, where steps from evaluation to mutation (steps 2–5) are iterated until a satisfactory solution is found or a predefined termination condition is met. This process of evolving the population continues over multiple generations. Termination conditions can vary, but common criteria include reaching a maximum number of generations, achieving a fitness threshold that defines a successful solution, or observing little to no improvement in fitness over several generations. Once one of these conditions is satisfied, the algorithm concludes, ideally with a high-quality solution to the optimization problem.

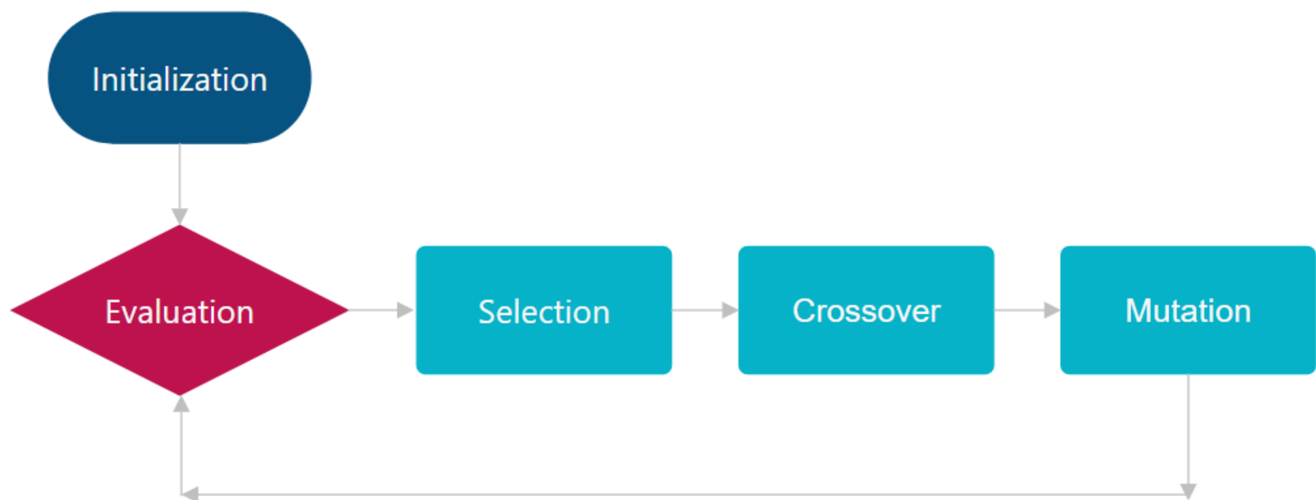


Figure 9: Genetic Algorithm Phases, Made by author

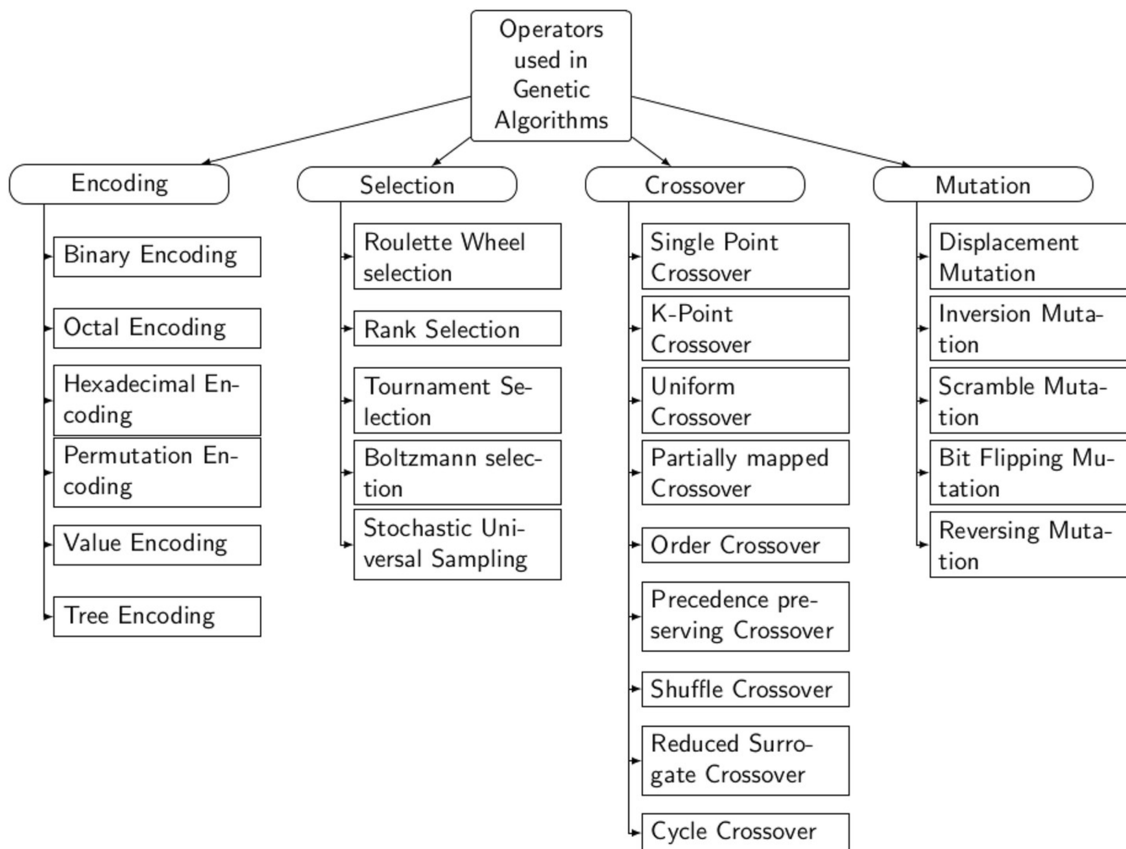


Figure 10: Operators in GAs, [13]

Summary of Key Components:

- **Population**: A set of candidate solutions.
- **Fitness Function or Evaluation**: Measures the quality of each solution.
- **Selection**: Chooses parents based on fitness to create the next generation.
- **Crossover**: Combines genetic material from two parents to create offspring.
- **Mutation**: Randomly alters genes to maintain diversity.
- **Termination or repeat**: The algorithm stops when a stopping condition is met (e.g., optimal solution found or maximum number of generations).

2.7 Advantages and Limitations of Genetic Algorithms

2.7.1 Advantages of Genetic Algorithms

Genetic Algorithms (GAs) offer a range of advantages that make them highly effective for solving diverse optimization problems. Their flexibility allows them to be applied to both discrete and

continuous problems, including multi-objective optimization. A key strength of GAs is their ability to balance exploration and exploitation: mutation and crossover facilitate the discovery of new solutions while simultaneously refining existing ones. This balance ensures efficient navigation through complex solution spaces. GAs are also noted for their robustness, as they can handle large state spaces and maintain performance in noisy environments, making them suitable for real-world applications. Additionally, they do not require gradient information, which makes them ideal for non-differentiable or discontinuous functions, unlike many traditional optimization methods. GAs also support parallelism, allowing multiple solutions to be evaluated simultaneously, enhancing computational efficiency. Over successive generations, GAs show improvement over time, evolving more refined solutions. Furthermore, their adaptability enables them to maintain performance even when problem conditions or environments change. [15]

2.7.2 Limitations of Genetic Algorithms

However, GAs are not without their limitations. A significant drawback is their high computational cost, as evaluating large populations across many generations can be resource-intensive. GAs can also suffer from premature convergence, where population diversity decreases too early, causing the algorithm to get stuck in suboptimal solutions or local optima. The success of GAs heavily relies on proper parameter tuning, including factors like population size, mutation rate, and crossover probability, which often requires trial and error. Additionally, GAs provide no guarantee of finding an optimal solution, and they may only arrive at a near-optimal result. Their slow convergence on flat or complex fitness landscapes can also prolong the search for a good solution. Moreover, the fitness function plays a crucial role, and a poorly designed one can misguide the algorithm. Some issues may also pose challenges in complex representation, where encoding solutions and designing genetic operations is difficult. Finally, GAs lack problem-specific knowledge, limiting their efficiency compared to specialized methods that incorporate domain expertise.

In summary, while genetic algorithms are flexible and powerful tools for optimization, their computational cost, sensitivity to parameters, and risk of premature convergence are important limitations to consider when choosing them for a particular application.

3. Introduction to Neural Networks
3.1 History of Neural Networks
3.1.1. Origins of Neural Networks
In 1936, Alan Turing invented a mathematical model of a universal machine, which later became known as a Turing Machine. [17] , [18] The Turing Machine laid the foundation for neural networks by introducing a formal model of computation. This framework allowed for the development of algorithms and logic essential to AI, including neural networks. Later on, in 1947, Turing likely delivered the earliest public lectures on computer intelligence, stating, "What we want is a machine that can learn from experience," and noting that the "possibility of letting the machine alter its own instructions provides the mechanism for this." In 1948, he introduced many core concepts of artificial intelligence in his report titled "Intelligent Machinery." [19]
The goal to simulate how neurons in the brain work is the origin of the neural network concept, which is a movement known as "connectionism." This concept was first put forth in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts, who suggested a basic electrical circuit that might simulate intelligent behavior. The neural network concept was first introduced by their work. In his book The Organization of Behavior, published in 1949, Donald Hebb developed this theory further and proposed that repeated usage strengthens brain connections, particularly those that are frequently engaged together. This served as a first step toward a mathematical understanding of how the brain works. [20], [21]

3.1.2 Developmental Phases of Neural Networks
Neural networks had a number of significant developmental stages starting in the 1940s. McCulloch and Pitts presented the first mathematical model of artificial neurons in the 1940s and 1950s. Further advancement was hindered, nevertheless, by the computing constraints of the era. Frank Rosenblatt developed a single-layer neural network called the perceptron in the 1960s and 1970s that could solve linearly separable problems. It was revolutionary, but it soon showed its limitations because it was unable to answer more complicated, non-linear problems. [20], [21]

3.1.3 Resurgence in the 1980s
The 1980s saw a revival in neural network research with the introduction of the backpropagation algorithm by Rumelhart, Hinton, and Williams. This technique allowed for the training of multi-layer neural networks, reigniting interest in "connectionism" and enabling more complex problem-solving. In the 1990s, neural networks began finding applications in areas like finance and image recognition. However, high computational costs and unfulfilled expectations led to a period of stagnation, sometimes referred to as the "AI winter," though progress continued in some areas. [20], [21]

3.1.4 Renaissance in the 2000s
Because of improvements in network topologies, increased accessibility to huge datasets, and advancements in processing power, neural networks had a resurgence by the 2000s. Deep learning emerged at this time, enabling multi-layered networks (also known as deep architectures) to handle progressively more difficult tasks. [20], [21]

3.1.5 Dominance in Modern Applications

From the 2010s to the present, neural networks have come to dominate the machine learning landscape, with deep learning architectures such as recurrent neural networks (RNNs) and convolutional neural networks (CNNs) transforming industries like healthcare, gaming, image recognition, and natural language processing. These innovations have driven remarkable breakthroughs and solidified neural networks as a core technology in artificial intelligence. [20], [21]

3.2 What is a Neural Network?

3.2.1 Definition of Neural Network

In the field of machine learning, a neural network—often referred to as an artificial neural network (ANN) or simply a neural net (NN)—is a computational model that draws inspiration from the architecture and functionality of biological neural networks found in the brains of animals.

Neural networks are designed to simulate the way human and animal brains process information, allowing machines to learn from data in a manner similar to how living organisms acquire knowledge through experience. A neural network's structure is usually made up of linked layers of nodes, or "neurons," each of which applies mathematical operations to the input data it receives. By altering the strengths of the connections, or weights, between neurons in these networks according to the input data and the intended output, these networks are able to recognize patterns, anticipate outcomes, and classify data. Neural networks are a valuable tool in the machine learning toolkit because of their versatility, which allows them to handle a wide range of tasks like image and speech recognition, natural language processing, and much more. [22]

3.2.2 Weights and Biases

Neural networks use fundamental parameters called weights and biases to assist the model identify patterns in the data and generate precise predictions.

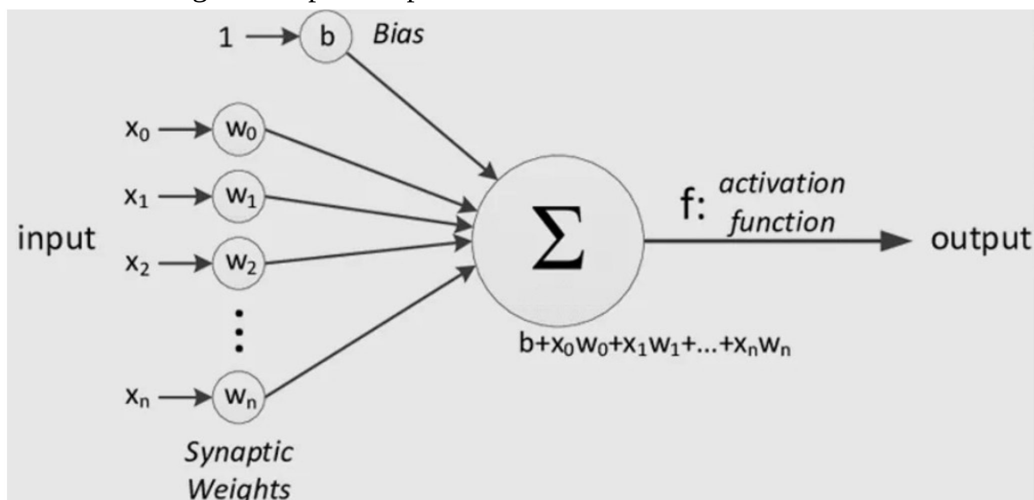


Figure 11: A simple perceptron, [23]

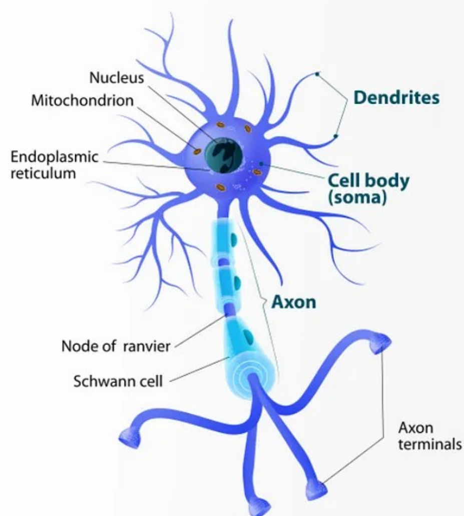
The strength of connections between neurons in various network levels is represented by weights. They ascertain the degree to which one neuron influences another. Data travels through the network during forward propagation, and the weight of each connection influences the information flow. The network must be trained with the weights adjusted, as doing so improves the network's pattern recognition capabilities. Biases are additional parameters that allow the model to shift the activation function. They assist in moving data across the network even in the event that the input values are zero and are independent of the main units. Biases direct data toward the ultimate output while ensuring that a neuron can still fire under particular circumstances. [23], [24]

In training, neural networks first use forward propagation to send data through layers and make predictions. Errors from these predictions are then used to adjust both weights and biases during backward propagation, where the network fine-tunes these connections to improve accuracy in future iterations.

3.2.2.1 Detailed explanations of the terms related to weights and biases

This section defines key concepts related to weights and biases in neural networks. Understanding these terms, such as neurons, layers, and hidden layers, is crucial for comprehending how neural networks process data and learn. Below are concise descriptions of each term.

- **Neuron**: In the context of neural networks, a neuron is a fundamental unit that processes individual data points or features. Each neuron receives input data, performs a computation (often involving weights and biases), and then passes an output to the next layer of neurons. Neurons work together to analyze and transform data as it moves through the network.



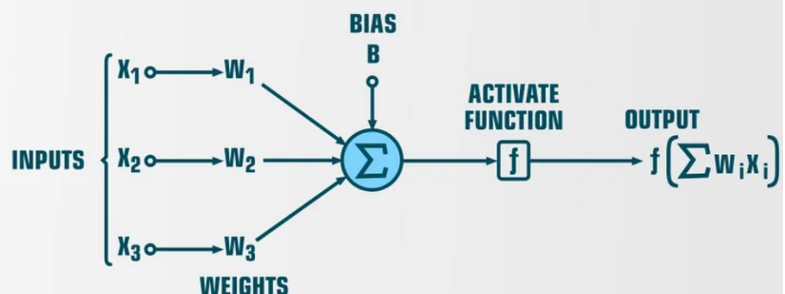


Figure 12: Structure of Typical Neuron and Artificial Neuron [23]

- **Layers**: Neural networks are structured in layers, each containing multiple neurons. The first layer, known as the input layer, consists of neurons that receive the raw data or features. These neurons connect to other layers, transmitting information forward. The process continues until the final set of neurons, called the output layer, produces the network's prediction or result. Weights govern how strongly neurons in one layer influence neurons in subsequent layers.
- **Hidden Layers**: Hidden layers lie between the input and output layers and are responsible for much of the network's computational power. Each hidden layer neuron takes in inputs from the previous layer, multiplies them by the corresponding weights, and passes the result through an activation function. This activation function introduces non-linearity, enabling the network to learn complex patterns. Hidden layers are crucial in deep learning, where networks often contain multiple hidden layers for advanced data processing and pattern recognition.

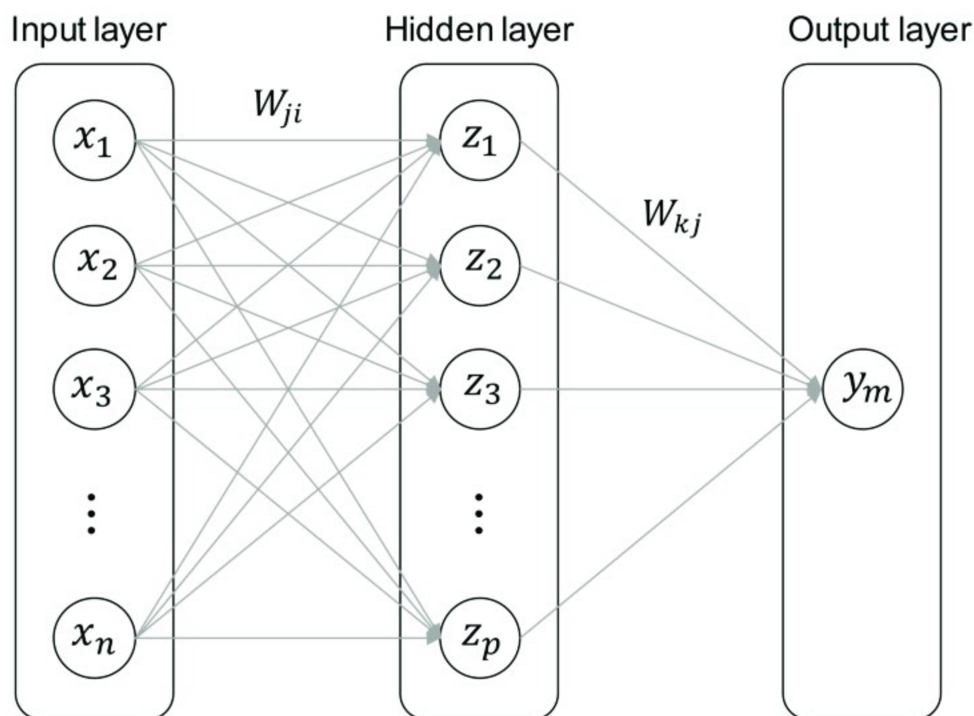


Figure 13: Basic structure of the layers of an Artificial Neural Network (ANN), [25]

3.3 Key Neural Network Concepts

Neural network fundamentals such as regularization, parameters, activation functions, and loss functions are described in the sections that follow. These components are essential to comprehending how neural networks handle information and raise the precision of their predictions.

3.3.1 Activation Function

An activation function determines whether a neuron should be activated or not based on its input. It introduces non-linearity to the model, allowing neural networks to capture complex patterns in data that linear models cannot handle. A neural network's capacity to address complicated issues would be constrained in the absence of an activation function, which would reduce it to a straightforward linear model. The model may learn complex correlations between input and output data thanks to the activation function, which changes the inputs. Typical activation functions in neural networks include Sigmoid, ReLU (Rectified Linear Unit), Leaky ReLU, and Tanh. [26], [27]

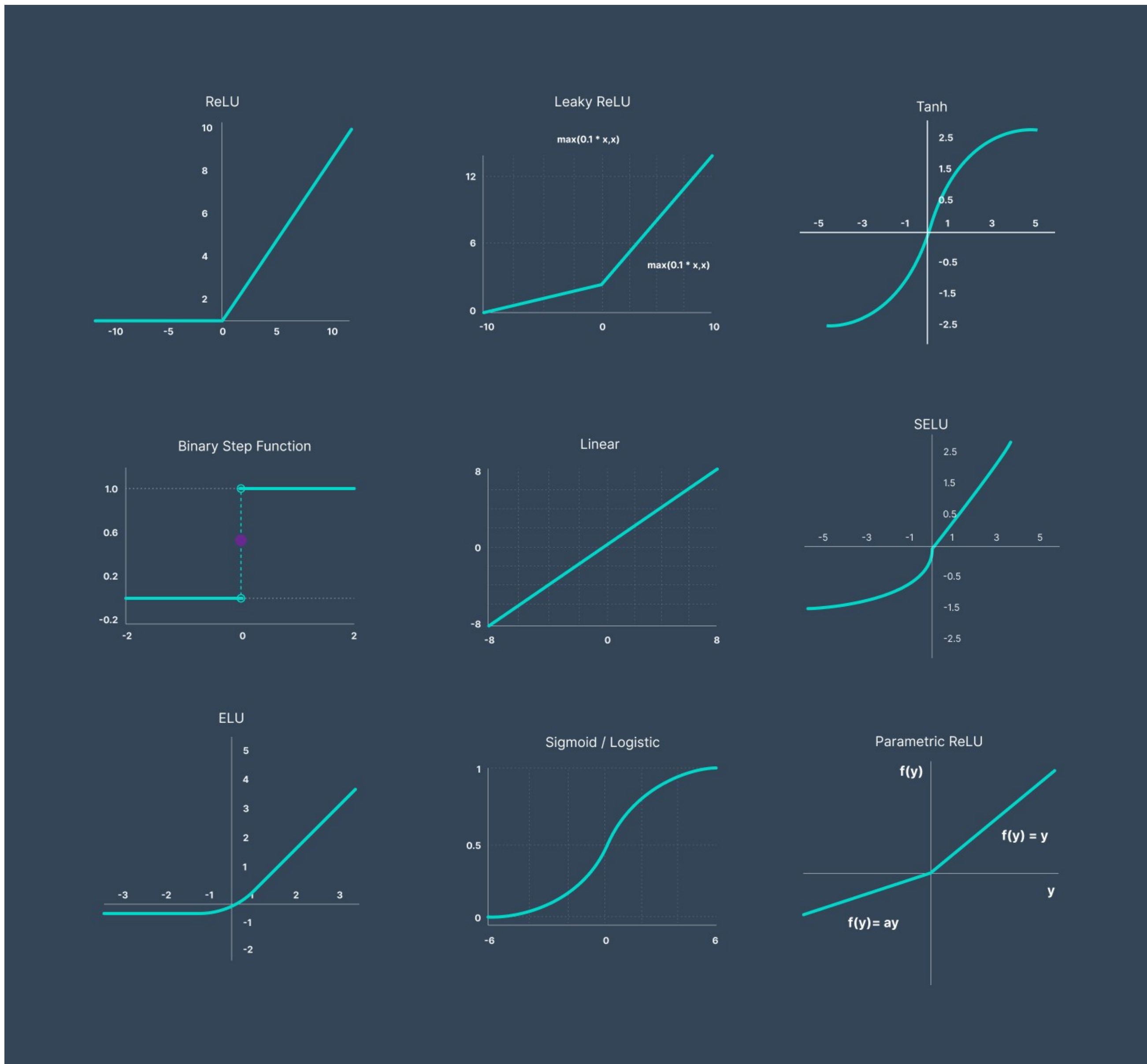


Figure 14: Activation Functions, [27]

The Sigmoid function is suitable for binary classification tasks as it compresses input values to a range between 0 and 1, acting as a smooth threshold that influences neuron activation. ReLU allows positive values to pass through unchanged while converting negative values to zero, promoting quick convergence during training and effectively addressing the vanishing gradient issue, making it particularly useful for deep networks. Leaky ReLU is similar to ReLU but permits a small, non-zero gradient for negative inputs, thus preventing neurons from becoming dormant due to zero gradients. Finally, Tanh produces output values between -1 and 1, providing both positive and negative activations, which is advantageous for scenarios where the data distribution near zero is important. Activation functions are crucial for introducing the necessary non-linearities that enable neural networks to approximate complex functions, making them highly effective for tasks such as image recognition, natural language processing, and more.

3.3.2 Loss Function

The loss function, also known as the error function, quantifies the difference between a neural network's predicted outputs and the actual target values. For example, in a regression task such as predicting car prices, it measures the error margin between predicted and actual prices, providing an indication of the model's accuracy. During training, algorithms like backpropagation use the gradient of the loss to adjust the model's parameters, guiding improvements in accuracy. Common loss functions include Mean Squared Error (MSE), which minimizes the squared differences between predicted and actual values for regression tasks, and Cross-Entropy Loss, which measures the dissimilarity between predicted probabilities and true labels for classification tasks. By minimizing the loss during training, the neural network enhances its predictive performance over time. [28]

3.3.3 Regularization

Regularization is a technique that prevents overfitting, where a neural network excels on training data but struggles with new data. It imposes constraints on the model's weights to keep them within a reasonable range, ensuring the model doesn't become overly complex. Common methods include L2 Regularization (Ridge), which penalizes large weights; L1 Regularization (Lasso), which encourages sparsity by driving some weights to zero; and Dropout, which randomly excludes a fraction of neurons during training to promote reliance on multiple features. Overall, regularization improves the model's robustness and generalization to real-world data. [29]

3.3.4 Parameters (Weights and Biases)

Weights and biases are crucial adjustable parameters in a neural network that define the strength and direction of connections between neurons. Weights determine how much influence an input neuron has on the activation of the next neuron, and during training, the algorithm adjusts these weights to minimize the loss function and enhance network accuracy. Biases allow the network to shift the activation function, helping it learn patterns that may not directly depend on the input values and providing greater flexibility in shaping decision boundaries. By adjusting weights and biases during training, the network can effectively capture complex patterns and relationships in the data. [30]

3.3.5 Bias Values

Bias values in neural networks act as a way to adjust the activation function, allowing neurons to activate even when inputs are zero. By adding a bias term, the network becomes more adaptable, as it can shift the activation function to better fit the data. This additional flexibility allows the model to generate more accurate predictions, especially when inputs do not perfectly correlate with outputs. Biases help in controlling the way inputs are processed, enhancing the network’s learning capability. [30]

3.4 Training Methods for Neural Networks

Training methods are crucial for enabling neural networks to learn from data effectively. The three main approaches are supervised learning, unsupervised learning, and reinforcement learning, each suited for different tasks based on the availability of labeled data. The following sections will briefly outline these methods and their specific applications. [31], [32]

3.4.1 Supervised Learning

In supervised learning, a "teacher" who is aware of the proper input-output pairings guides the neural network during its learning process. The network produces a result for each input without taking outside variables into account. The expected result given by the teacher is then compared to the predicted output, and if there is a discrepancy, an error signal is generated. To lessen the discrepancy between the expected and actual outputs, the network's parameters (weights and biases) are iteratively adjusted using this error. The procedure keeps on until the accuracy of the network's performance is at a level that is satisfactory. When labeled data is available, activities like regression and classification are frequently handled using this technique.

3.4.2 Unsupervised Learning

In unsupervised learning, the network functions without output data that has been labeled. Finding hidden patterns or structures in the input data (X) without the aid of a "teacher" or outside advisor is the aim. Understanding the relationships within the data is the network's primary goal, as there are no predetermined proper outputs. Rather than aiming for a certain result, the model finds patterns in the dataset, such as clusters or correlations. Unsupervised learning is mostly utilized in tasks like association (identifying associations between variables) and clustering (grouping similar data), whereas supervised learning is used for problems like regression and classification.

3.4.3 Reinforcement Learning

Reinforcement learning allows a neural network to learn by interacting with its environment and receiving feedback in the form of rewards or penalties. The network’s goal is to develop a policy or strategy that maximizes cumulative rewards over time. It continuously improves by taking actions in the environment, observing the outcomes, and adjusting its actions based on the feedback. Reinforcement learning is widely used in scenarios that require decision-making, such as gaming, robotics, and autonomous systems, where the objective is to optimize long-term performance.

3.5 Types of Neural Networks

There are several different types of neural networks, each suited for specific tasks depending on the architecture and data being processed. Here are the most commonly used types: [32], [33]

3.5.1 Feedforward Neural Networks (FNN)

Feedforward neural networks represent one of the simplest forms of artificial neural networks. In this architecture, data flows in a single direction, from the input layer through any hidden layers to the output layer, without any cycles or feedback loops. The absence of feedback makes this network straightforward and efficient for tasks like regression analysis and pattern recognition. Feedforward networks are typically used when the relationships between inputs and outputs are relatively straightforward.

3.5.2 Multilayer Perceptron (MLP)

A more sophisticated kind of feedforward network is the multilayer perceptron (MLP), which consists of an input layer, one or more hidden layers, and an output layer. MLPs are unique in that they may represent intricate interactions between inputs and outputs because they use nonlinear activation functions. MLPs are frequently used for jobs requiring more complex function approximation and pattern recognition.

3.5.3 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized for processing grid-like data structures, such as images. They use convolutional layers that automatically learn features from the input data, progressively capturing patterns at different levels of complexity, from simple edges to intricate shapes. CNNs have revolutionized computer vision, making them essential for tasks like image classification, object detection, and facial recognition. Their ability to efficiently process visual data has made them a key component in many modern image-based AI applications.

3.5.4 Recurrent Neural Networks (RNNs)

When processing sequential data—such as time series data or spoken language—recurrent neural networks, or RNNs, are optimized to handle context-sensitive scenarios. Because they have feedback loops, which enable information to persist across time steps, RNNs are not like feedforward networks. Because of this, RNNs are perfect for applications such as time-dependent data processing, speech recognition, and language modeling. However, because of the vanishing gradient issue, conventional RNNs may have trouble with long-term dependencies.

3.5.6 Long Short-Term Memory (LSTM)

LSTM is a specific type of RNN designed to overcome the limitations of traditional RNNs, particularly the vanishing gradient problem. LSTMs use special memory cells and gates that can selectively retain or discard information over long sequences. This makes them particularly effective in handling long-term dependencies and complex temporal patterns. LSTMs are widely used in tasks such as natural language processing, speech synthesis, and time series forecasting, where maintaining contextual information over long periods is critical.

Each of these neural network types has its own strengths, making them suitable for various tasks depending on the complexity and nature of the data.

3.6 Forward Propagation and backpropagation in NNs

Forward propagation and backpropagation are essential processes in neural networks. Forward propagation involves the flow of input data through the network, leading to predictions or classifications. In contrast, backpropagation adjusts the network's weights and biases based on the prediction errors, facilitating learning. Together, these processes enable neural networks to process information and improve accuracy through continuous feedback. The following sections will explore each process in detail. [34], [35]

3.6.1 Forward propagation

A neural network's data processing sequence begins with forward propagation. In this stage, new data is added to the network and is routed across a number of interconnected levels. Within these layers, every neuron performs particular mathematical operations on the incoming data, changing it and sending the information that has been processed to layers above. This process keeps going until the data gets to the last output layer, where the network produces its classifications or predictions. Forward propagation is a linear, unidirectional process in which data moves smoothly from input to output in the absence of feedback loops.

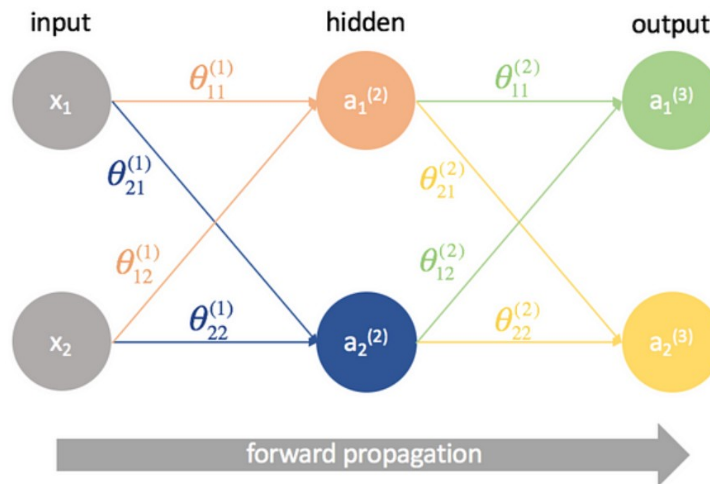


Figure 15: Forward propagation, [34]

3.6.2 Back propagation

Conversely, back propagation functions as the network's learning process. The neural network evaluates its performance by comparing the output to the predetermined target or desired outcome when forward propagation is finished and an output is generated. Critical feedback is provided by the difference, sometimes known as the error or loss, between the output that was produced and the intended output. By using an inverted process that goes from the output layer back to the input layer, this error is used to modify the weights and biases of the network. By improving the network's

parameters iteratively, backpropagation seeks to reduce this mistake and improve the network's accuracy and predictive capacity.

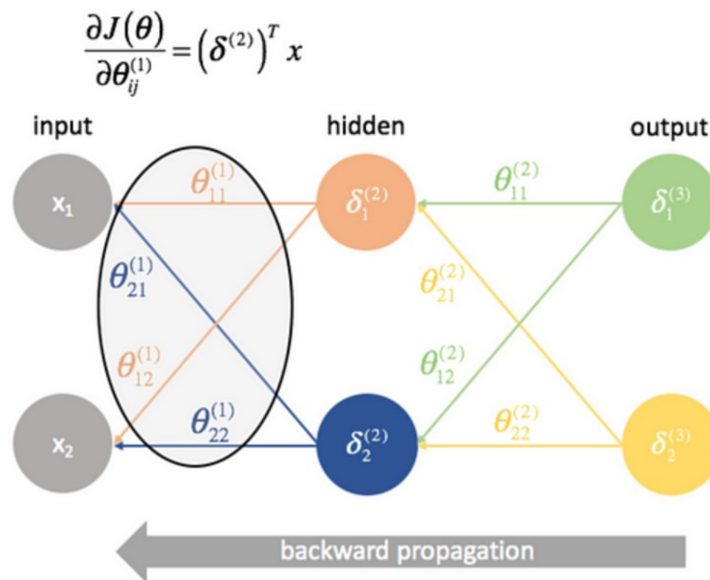


Figure 16: Backpropagation, [34]

3.6.3 Comparing Forward and backpropagation

Back propagation aims to enhance the network's performance by learning from the differences between expected and actual outputs, whereas forward propagation concentrates on processing data and generating outputs. Neural networks require both of these processes to function properly; they cooperate to make sure the network not only processes information but also learns from its errors. While back propagation makes it easier for the network to continuously develop based on user feedback, forward propagation lays the groundwork for producing outputs.

Both forward and backpropagation have inherent difficulties despite their efficacy. Processing times might increase due to forward propagation's computing demands, which are particularly noticeable in intricate neural network topologies with many of layers and neurons. A unique series of problems affect back propagation, most notably the vanishing gradient problem, in which gradients, which are required to update weights, get unnecessarily small during training. This may impede meaningful changes to the network's parameters, which in turn may impair the model's capacity to learn and to converge.

4. Introduction to Convolutional Neural Networks (CNNs)

4.1 Definition of a CNN

A **Convolutional Neural Network (CNN** or **ConvNet)**, is a type of deep learning architecture specifically designed to process structured grid-like data, such as images. It plays a pivotal role in computer vision, a field within artificial intelligence that enables computers to analyze and interpret visual data, such as images and videos, in a way that mimics human vision.

While traditional artificial neural networks (ANNs) perform well across a range of tasks like image recognition, speech processing, and text analysis, CNNs are particularly powerful in image-based applications. They are the go-to architecture for complex tasks like image classification, object detection, and segmentation. The specialized structure of CNNs makes them more efficient and accurate in handling visual data compared to standard neural networks. [36], [37]

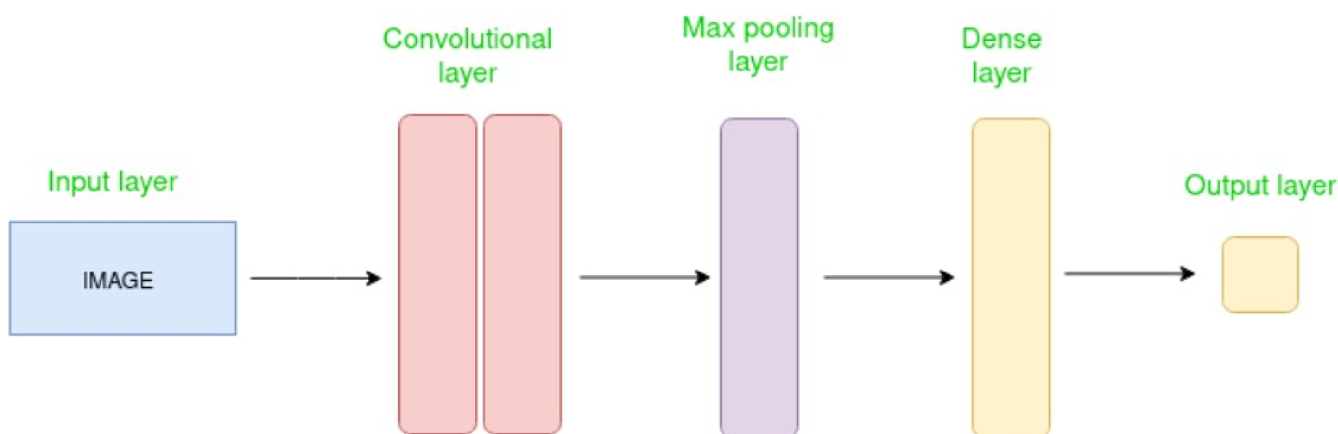


Figure 17: Basic structure of a CNN [36]

Data passes through several layers of a CNN, each of which is intended to gradually extract useful features from unprocessed picture data. The network recognizes increasingly sophisticated structures, such as shapes, objects, and even faces, as input moves through successive layers. Convolutional layers use filters to identify low-level patterns like edges, corners, and textures. CNNs are able to identify and categorize things with remarkably high accuracy thanks to hierarchical feature extraction.

CNNs are used in many different fields, such as autonomous cars, where they aid in the detection of other vehicles and pedestrians, security systems, which employ CNNs for facial recognition, and medical imaging, where they aid in the identification of diseases from X-rays or MRIs. Developments in augmented reality, gaming, and robotics also depend on CNNs.

Depending on the type of data, different neural network types have distinct uses in the larger field of machine learning. For instance, Recurrent Neural Networks (RNNs) or more sophisticated models like Long Short-Term Memory (LSTM) networks are employed for tasks involving sequential input, such as language modeling. On the other hand, because convolutions and pooling layers allow CNNs to

extract spatial hierarchies from the data, they perform better than classic ANNs when working with static images. [36], [37]

4.2 Core Elements of a Convolutional Neural Network

A Convolutional Neural Network (CNN) consists of four key components that work together to process and analyze images, helping the network recognize patterns and features in a way that resembles how the human brain interprets visual information. Next, we'll see how CNNs learn using these parts. [38],[39], [40], [41]

4.2.1 Convolutional Layers

The convolutional layer is the first crucial component in a CNN. It applies a series of filters or kernels to the input image, allowing the network to detect essential features such as edges, textures, and colors. These filters slide over the input, performing element-wise multiplications, which capture different patterns at various spatial locations. In the case of handwritten digit classification, the convolutional layers may detect curves, lines, and corners that form parts of numbers, like the curve in "3" or the vertical line in "1." These extracted features are then passed to the next layers for further processing.

4.2.2 Rectified Linear Unit (ReLU)

After the convolutional layer, the ReLU activation function is applied to the feature maps. This function introduces non-linearity by converting all negative pixel values to zero while keeping positive values unchanged. This non-linearity is essential because, without it, the CNN would only be able to capture linear relationships, which limits its ability to recognize complex patterns. By introducing non-linearities, ReLU enables the network to learn intricate features, such as distinguishing between different shapes in a handwritten digit.

4.2.3 Pooling Layers

Pooling layers, often called subsampling or down-sampling layers, reduce the spatial dimensions of the feature maps. This helps to decrease the computational load and the number of parameters in the network, making it more efficient. The most common type is max pooling, which takes the maximum value from a small region of the feature map, preserving the most significant information. In the handwritten digit example, pooling layers would condense the feature maps, allowing the network to retain essential features (like the overall shape of the number) while discarding less important details, such as noise or minor variations in the strokes.

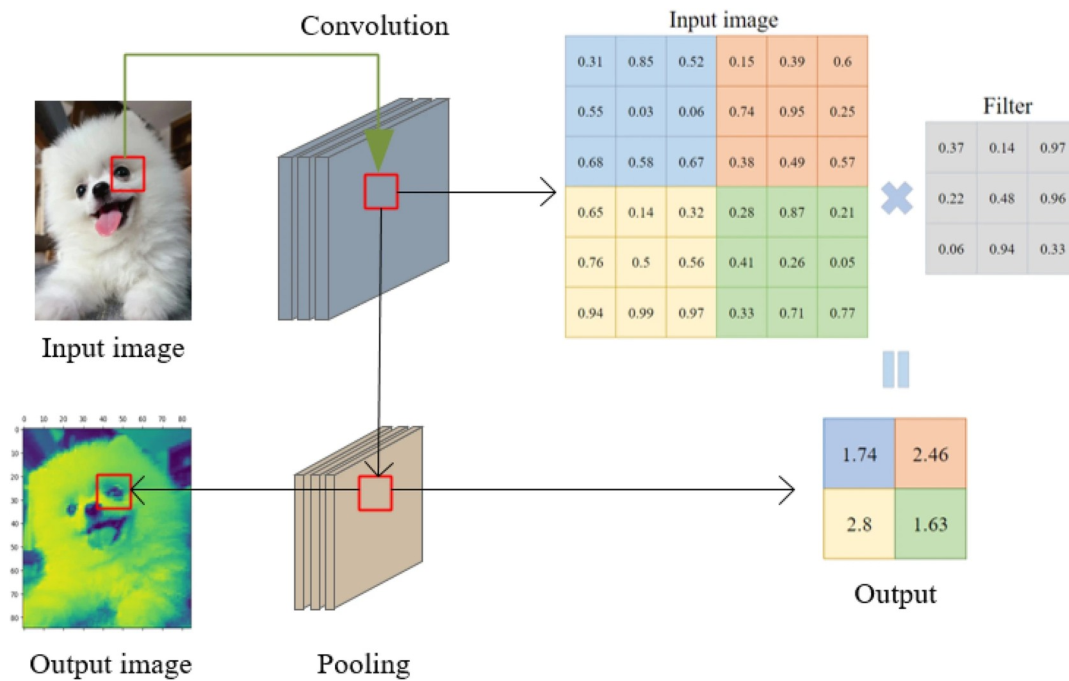


Figure 18: Pooling method for convolutional neural networks, [39]

Some of the known pooling layers are Max Pooling, Average Pooling, Mixed Pooling, and Stochastic Pooling Max Pooling is a down-sampling technique used in CNNs to reduce the spatial dimensions of the input feature map while retaining important information. It chooses the maximum value from each zone after breaking the input into smaller halves. This preserves important properties while lowering the amount of computation, parameters, and overfitting risks. For example, when applying max pooling with a 2x2 filter and a stride of 2, the filter will move over the feature map in non-overlapping 2x2 sections, taking the maximum value from each section. [41]

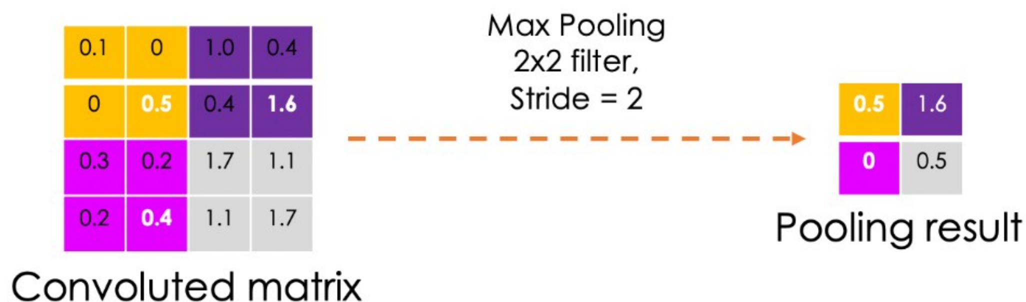


Figure 19: Max Pooling example, [41]

4.2.4 Fully Connected Layers

Finally, the fully connected layer connects every neuron from the previous layer to every neuron in the current layer, integrating the extracted features to make predictions. This layer is responsible for combining all the features learned in the earlier stages and assigning a probability score to each class label. For example, in digit classification, the fully connected layer takes the features from the convolutional and pooling layers and produces a final prediction, such as determining that the input image corresponds to the digit "7."

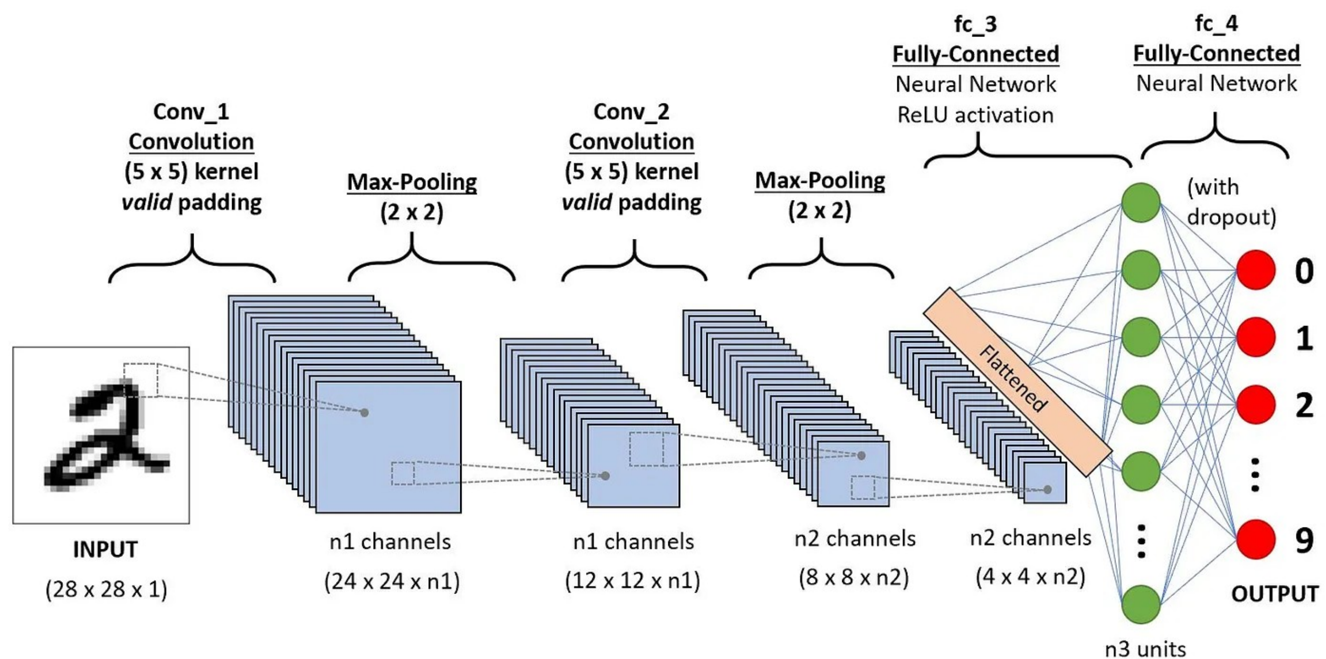


Figure 20: Example of a CNN sequence to classify handwritten digits, [40]

4.2.5 Python Code for a CNN Model

Example Python Code for a CNN Model Featuring 5x5 Kernels and Max Pooling Layers:

```
#Importing libraries
import tensorflow as tf
from tensorflow.keras import layers, models

# Defining the CNN model
model = models.Sequential()
# Convolutional Layer with 5x5 Kernel
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(28, 28, 1)))
```

```
# Max Pooling Layer
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
# Second Convolutional Layer with 5x5 Kernel
model.add(layers.Conv2D(64, (5, 5), activation='relu'))
# Max Pooling Layer
model.add(layers.MaxPooling2D(pool_size=(2, 2)))

# Flattening Layer
model.add(layers.Flatten())
# Fully Connected Layer
model.add(layers.Dense(128, activation='relu'))

# Output Layer (For example for 10 classes for digit classification)
model.add(layers.Dense(10, activation='softmax'))

# Compiling the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Display the model summary
model.summary()
```

4.3 Underfitting and Overfitting in convolutional neural network (CNNs)

Comprehending underfitting and overfitting is crucial for convolutional neural networks (CNNs), as both play a significant role in a model's generalization capability. Underfitting results from a lack of complexity in the model, whereas overfitting happens when the model captures noise rather than important patterns. This section examines the features, causes, and remedies for both issues in CNNs. [42], [43]

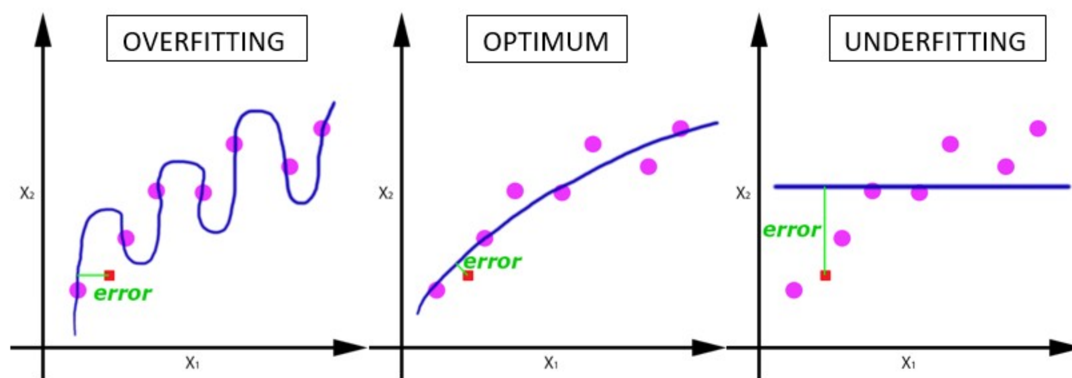


Figure 21: Overfitting & Underfitting example, [43]

4.3.1 Underfitting in CNNs

4.3.1.1 Exploring Underfitting in CNNs

Underfitting occurs when a convolutional neural network (CNN) is too simplistic to capture the complexities of the training data, leading to poor performance on both training and testing datasets. In

this situation, the model fails to learn effectively, resulting in inaccurate predictions, especially for new and unseen examples. Several factors contribute to underfitting, including overly simple models that lack the complexity necessary to represent underlying data patterns. Additionally, if the features derived from the input data are insufficient or unimportant, they may fail to adequately depict the variables affecting the target variable. Limited training data can also hinder the model's learning capabilities, as a small dataset may not provide enough information for the model to generalize effectively. Furthermore, excessive regularization can constrain the model, preventing it from adequately representing the subtleties of the data.

4.3.1.2 Strategies to Overcome Underfitting

Models that experience underfitting typically exhibit high bias and low variance. This means they make strong assumptions about the data, leading to systematic errors in predictions. To address underfitting, several strategies can be employed. Increasing model complexity is one approach, as it allows the network to better capture data nuances. Enhancing feature representation through feature engineering can also improve the model's ability to learn. Lastly, ensuring that the model has sufficient training duration is critical for effective learning. By implementing these strategies, the model's performance can be significantly improved.

4.3.2 Overfitting in CNNs

4.3.2.1 Causes of Overfitting in CNNs

Overfitting occurs when a convolutional neural network (CNN) learns the noise and inaccuracies in the training dataset instead of the underlying patterns, resulting in poor generalization to unseen data. This problem is characterized by high variance and low bias, meaning the model becomes overly complex and overly sensitive to the training data. Several factors contribute to overfitting, including excessive model complexity, where highly flexible models capture too many details of the training data; small training datasets, which can lead the model to learn noise instead of meaningful patterns; and a lack of regularization, allowing the model to fit the training data too closely, including irrelevant features.

4.3.2.2 Strategies for Mitigating Overfitting

To mitigate overfitting, various techniques can be employed. Improving the quality of training data by focusing on meaningful patterns can help reduce the risk of fitting noise. Increasing the volume of training data enhances the model’s ability to generalize. Reducing model complexity through simplification can prevent the model from learning irrelevant details. Implementing early stopping, which involves monitoring the loss during training and halting when it starts to increase, can also prevent overfitting. Additionally, applying regularization techniques like Ridge and Lasso can help constrain the model's complexity. Lastly, using dropout layers in neural networks randomly deactivates a subset of neurons during training, reducing reliance on any single neuron and promoting better generalization.

Understanding and addressing underfitting and overfitting are crucial for building effective CNNs. Striking a balance between model complexity and generalization ensures that CNNs can learn effectively from the training data while maintaining the ability to perform well on new, unseen data.

5. Overview of Generative Adversarial Networks (GANs)

5.1 Introduction to GANs

Generative Adversarial Networks (GANs) are a potent class of machine learning models that use a framework of competition between two neural networks—a discriminator and a generator—to produce realistic data. GANs operate on a zero-sum game, where the success of one network comes at the price of the other, and they make use of unsupervised learning. Their popularity stems from their ability to generate new realistic data, including human-face photos, which they may use for tasks like style transfer, image production, and even deepfakes. [44], [45]

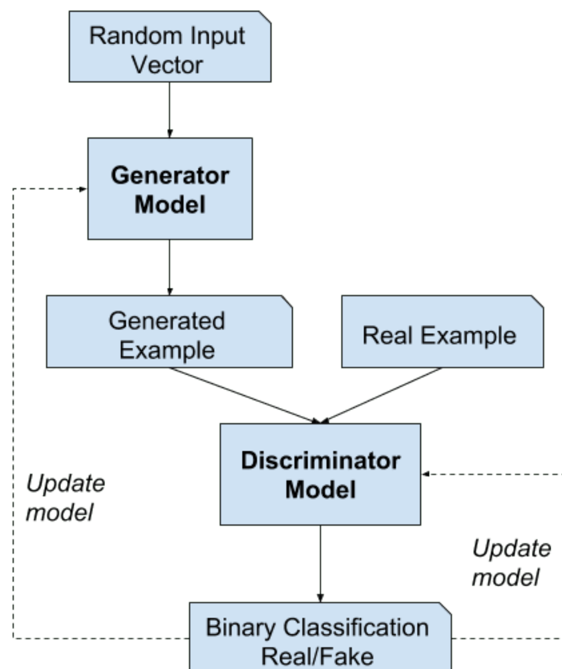


Figure 22: Basic structure of GAN, [45]

5.2 The main components of a GAN

Generative Adversarial Networks (GANs) comprise two essential components: the generator and the discriminator. The generator produces synthetic data, while the discriminator assesses its authenticity against real data. This competitive dynamic fosters continuous improvement in both networks, enhancing GANs' effectiveness in tasks like image generation and data augmentation. The following sections will delve into the specific roles of the generator and discriminator in a GAN.

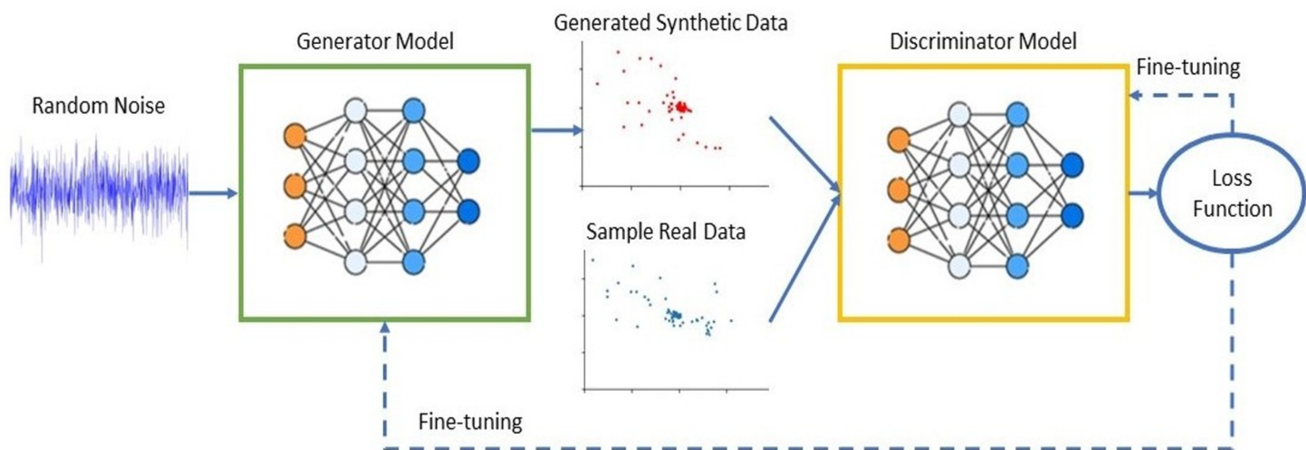


Figure 23: General workflow of GAN architecture, [46]

5.2.1 The Role of the Generator in GANs

In Generative Adversarial Networks (GANs), the generator is essential for creating synthetic data that closely resembles real training data. It takes random noise as input and transforms it into outputs, such as images or text, that aim to appear authentic. Typically structured as a convolutional neural network (CNN), the generator refines its abilities through iterative training, learning from feedback provided by the discriminator—a separate network that assesses the realism of the generated samples. As the generator evolves, it becomes increasingly skilled at producing high-quality outputs that capture the underlying data distribution. This dynamic between the generator and discriminator enables GANs to generate high-fidelity samples, making them valuable tools in applications like image synthesis and data augmentation.

5.2.1.1 A basic generator model example in Python code

```
#Defining the generator model
def build_generator():
    model = Sequential()
    model.add(Dense(128, input_dim=100)) #100 is the noise vector size
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(784, activation='tanh')) #Output a 28x28 image(flattened)
    model.add(Reshape((28, 28, 1))) #Reshape into 28x28x1 image
    return model
```

5.2.2 The Role of the Discriminator in GANs

In Generative Adversarial Networks (GANs), the discriminator is essential for distinguishing between real and generated data. Acting as a deconvolutional neural network, its main goal is to classify incoming samples as either authentic—coming from the training dataset—or fake—produced by the generator. As the discriminator becomes more adept at identifying subtle differences between genuine and synthetic data, it challenges the generator to create increasingly convincing outputs. This adversarial relationship promotes a dynamic learning environment, where the generator continuously improves its ability to produce realistic samples, while the discriminator enhances its detection capabilities. Ultimately, this ongoing competition not only boosts the performance of both networks but also allows GANs to generate high-quality outputs for applications like image synthesis and data augmentation.

5.2.2.1 A basic generator model example in Python code

```
#Defining the discriminator model
def build_discriminator():
    model = Sequential()
    model.add(Flatten(input_shape=(28, 28, 1)))
    model.add(Dense(128))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(1, activation= 'sigmoid')) #Binary classification(real/fake)
    return model
```

5.2 How Do GANs work?

Two neural networks that participate in an adversarial process—the discriminator and the generator—make up a generative adversarial network, or GAN. The discriminator's job is to identify real from fake data, while the generator's is to produce data that looks like the real training set. The generator initially generates outputs of poor quality, but when the two networks interact and create a feedback loop, both networks get better. When the discriminator properly detects bogus data, the generator is penalized and compelled to make improvements, which eventually teaches it to generate outputs that are more realistic. The discriminator gets more adept at spotting fakes as time goes on, while the generator produces credible, high-quality data. [47], [48]

5.2.1 Training the generator and discriminator

In a Generative Adversarial Network (GAN), training the discriminator and generator is a dynamic process where both networks gain knowledge and get better through competition. The discriminator, a binary classifier, is trained to discern between the generator's bogus data and actual data from the dataset. This training procedure begins by constructing a dataset with both real and artificial samples. After that, backpropagation is used to train the discriminator so that it can recognize bogus data with greater accuracy. Simultaneously, the generator is optimized to provide fake data that can trick the discriminator. As part of its training process, the generator creates fictitious samples, feeds them to the discriminator, computes the loss—a measure of how successfully the generated data tricked the discriminator—and uses backpropagation to tweak its weights in order to produce better results.

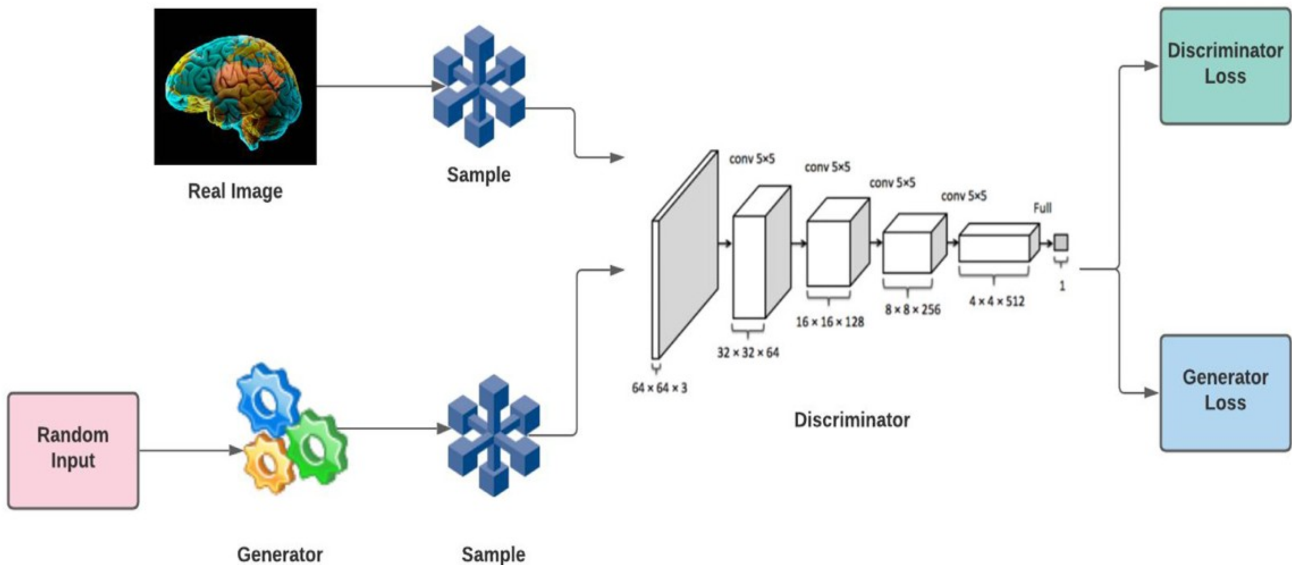


Figure 24: Block diagram of GAN, [49]

An adversarial training method is used to train both networks simultaneously. In this method, batches of actual and fake data are created, random noise vectors are generated, and the discriminator is trained on these batches. Next, the generator attempts to trick the discriminator by producing increasingly realistic-looking phony data. The generator gets stronger at creating realistic data as a result of this adversarial training, while the discriminator gets better at identifying fakes. Eventually, incredibly realistic synthetic data that can be challenging to identify from genuine data are produced by GANs thanks to this competitive process. Because GANs constantly adapt to each other, training them can be difficult, but when done right, the output is striking and lifelike, especially for tasks like image production. [44], [48]

5.2.2 Mathematical Explanation of Training

The training process can be described through the following mathematical expressions. In the actual training the min-max problem is solved by alternately updating the discriminator D(s) and the

generator D(z). Since the discriminator D(s)D(s)D(s) represents the probability that a sample sss originates from the data distribution, it can be formulated as follows:

$$D(\mathbf{s}) = \frac{p(\mathbf{s})}{p(\mathbf{s}) + p_{\text{model}}(\mathbf{s})}$$

Then, when we match the data distribution $\mathbf{s} \sim p(\mathbf{s})$ and the distribution of generated samples by G, it means that we should minimize the dissimilarity between the two distributions. It is common to use **Jensen-Shannon Divergence** D_{JS} to measure the dissimilarity between distributions[3].

The D_{JS} of $p_{\text{model}}(\mathbf{s})$ and $p(\mathbf{s})$ can be written as follows by using $D(\mathbf{s})$:

$$\begin{aligned} 2D_{\text{JS}} =& \; D_{\text{KL}}(p(\mathbf{s})||\bar{p}(\mathbf{s})) + D_{\text{KL}}(p_{\text{model}}(\mathbf{s})||\bar{p}(\mathbf{s})) \\ =& \mathbb{E}_{p(\mathbf{s})}\left[\log \frac{2p(\mathbf{s})}{p(\mathbf{s}) + p_{\text{model}}(\mathbf{s})}\right] + \mathbb{E}_{p_{\text{model}}}\left[\log \frac{2p_{\text{model}}(\mathbf{s})}{p(\mathbf{s}) + p_{\text{model}}(\mathbf{s})}\right] \\ =& \; \mathbb{E}_{p(\mathbf{s})} \log D(\mathbf{s}) + \mathbb{E}_{p_{\text{model}}} \log(1 - D(\mathbf{s})) + \log 4 \\ =& \; \mathbb{E}_{p(\mathbf{s})} \log D(\mathbf{s}) + \mathbb{E}_{p_{\mathbf{z}}} \log(1 - D(G(\mathbf{z}))) + \log 4 \end{aligned}$$

where $\bar{p}(\mathbf{s}) = \frac{p(\mathbf{s}) + p_{\text{model}}(\mathbf{s})}{2}$. The D_{JS} will be maximized by the discriminator D and minimized by the generator G, namely, p_{model}. And the distribution $p_{model}(\mathbf{s})$ generated by $G(\mathbf{s})$ can match the data distribution $p(\mathbf{s})$.

$$\min_G \max_D \mathbb{E}_{p(\mathbf{s})} \log D(\mathbf{s}) + \mathbb{E}_{p_{\mathbf{z}}} \log(1 - D(G(\mathbf{z})))$$

Figure 25: Mathematical Explanation of Training, [50]

5.3 Applications of GANs

The ability of Generative Adversarial Networks (GANs) to generate and improve data has revolutionized many fields. GANs are used in image processing to generate highly realistic synthetic images, including human faces, and to convert images between other domains, such as converting drawings into detailed photos or transferring styles. By producing super-resolution images from low-resolution inputs, they also significantly contribute to the improvement of image resolution. GANs are utilized in the entertainment and creative industries to generate art, music, and video material. They are frequently employed for jobs like virtual avatars, animation, and deepfake production.

GANs are revolutionizing the healthcare industry by producing artificial medical images that help train machine learning models in situations when real data is hard to come by. By boosting image quality in medical imaging applications like CT and MRI scans, they contribute to increased diagnostic accuracy. GANs are also useful in drug development since they produce novel molecular structures for prospective drugs.

Beyond picture and medical applications, GANs are used in text-to-image synthesis (converting verbal descriptions into images), video prediction (predicting next frames in video sequences), and 3D object development. They are also used in data augmentation, which creates more artificial training data to improve model performance, and anomaly detection, which creates normal data patterns to find outliers. All things considered, GANs find many uses in domains ranging from the arts to science and business. [51], [52]

5.4 Types of GANs

There are several varieties of Generative Adversarial Networks(GANs), each intended to handle a particular generative modeling problem. The most common design, known as the Vanilla GAN, uses a discriminator to discern between genuine and false input, while a generator produces fake data. Deep Convolutional GAN (DCGANs) are especially useful for picture data since they use Convolutional Neural Networks (CNNs) to improve the quality of the images that are generated. While Wasserstein GANs (WGANs) change the loss function to overcome typical training concerns, Conditional GANs (cGANs) generate data based on specified labels or information. CycleGANs can learn without paired input, making them well-suited for image-to-image translation challenges. Last but not least, StyleGANs offer the capacity to regulate particular style attributes in produced images, enabling higher visual versatility and high-resolution outputs. Because of its distinct benefits, each kind of GAN can be used in a variety of generative modeling applications. [51], [52]

6. Deep Convolutional GANs (DCGANs)

6.1 Introduction to DCGANs

Deep Convolutional GANs (DCGANs) mark a significant advancement in the architecture of Generative Adversarial Networks, particularly for image generation tasks. Unlike traditional GANs that utilize fully connected layers and multi-layer perceptrons, DCGANs employ deep convolutional neural networks (CNNs) for both the generator and discriminator networks. This architectural shift enhances the model's ability to capture spatial features more effectively, resulting in higher-quality images that exhibit greater texture and detail. [53], [54]

6.1.1 Architectural Innovations

DCGANs feature key architectural modifications, such as the use of fractional-strided convolutions in the generator and strided convolutions in the discriminator, which replace conventional pooling layers. Additionally, both networks incorporate batch normalization to stabilize training. The generator employs the ReLU activation function, except for the output layer, where the tanh function is used. In contrast, the discriminator utilizes LeakyReLU. This design enables deeper architectures without the risk of overfitting that often accompanies fully connected layers.

6.1.2 Image Generation Process

The image generation process in DCGANs begins by feeding a noise vector into the generator, which then up-samples this vector to create an image. Simultaneously, the discriminator uses convolutional layers to down-sample the images, determining whether they are real or generated. Through adversarial training, the discriminator sharpens its ability to distinguish between authentic and fake images, while the generator improves its capacity for producing realistic outputs.

6.1.3 Challenges and Solutions

While DCGANs excel at generating high-quality images, they are not without challenges, such as instability during training and mode collapse, where the generator produces a limited range of outputs. These issues can be mitigated through various strategies, including regularization techniques, architectural adjustments, and modifications to the loss functions.

6.1.4 Practical Application

An example of DCGAN's capabilities was showcased during the ICLR presentation on LSUN scene generation. Here, the architecture demonstrated its ability to project a 100-dimensional noise vector into a 7x7x256 tensor, which is then progressively convolved to generate 28x28x1 MNIST digits. For LSUN scene modeling, the DCGAN generator similarly transforms the 100-dimensional noise vector into a small spatial convolutional representation with multiple feature maps, followed by four

fractional-strided convolutions—often incorrectly referred to as deconvolutions—to upscale the representation into a 64x64 pixel image. Notably, the architecture avoids fully connected or pooling layers, further emphasizing its unique design.

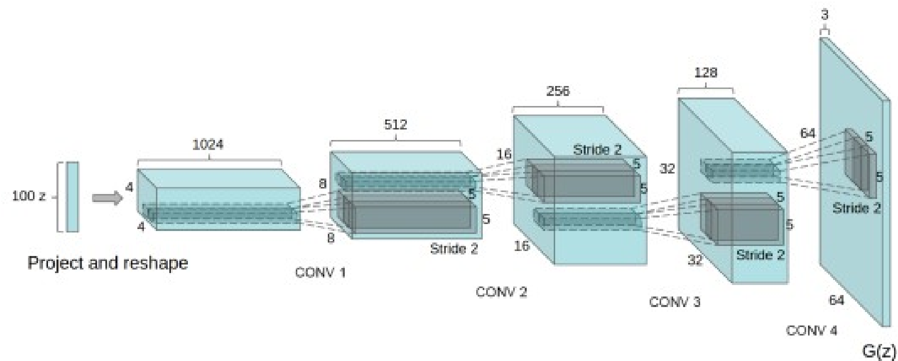


Figure 26: DCGAN architecture, [55]

6.1.5 Algorithm for Minibatch Stochastic Gradient Descent in GANs

The following outlines Algorithm 1, detailing the training process for GANs using minibatch stochastic gradient descent. The discriminator is updated over multiple steps (determined by the hyperparameter k), followed by an update to the generator. Gradients for both networks are computed and adjusted iteratively to optimize performance. The algorithm leverages momentum-based gradient updates to enhance training efficiency. In the experiments, $k = 1$ was selected as the least computationally expensive choice.

for number of training iterations **do**
for k steps **do**
• Sample minibatch of m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
• Sample minibatch of m examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distributior $p_{\text{data}}(x)$.
• Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right].$$

end for
• Sample minibatch of m noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
• Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figure 27: Minibatch stochastic gradient descent training of GANs, [56]

6.2 Key Differences Between GANs and DCGANs

Generative Adversarial Networks (GANs) encompass a broad category of models designed for generating new data, whereas Deep Convolutional GANs (DCGANs) represent a specialized subset tailored for image generation tasks. The primary distinctions between these two types of models stem from differences in their architecture, input handling, and performance in image-related applications. [57], [58]

6.2.1 Architecture

In traditional GANs, both the generator and discriminator networks rely on fully connected, or dense, layers to process data. This architecture, while flexible, often struggles to capture spatial relationships crucial in tasks like image synthesis. In contrast, DCGANs replace these dense layers with convolutional layers. The generator employs transposed, or fractionally-strided, convolutions, while the discriminator utilizes standard convolutions. This modification allows DCGANs to better capture spatial features, resulting in the generation of more realistic images with enhanced texture and detail.

6.2.2 Image Generation

The ability to generate high-quality images is another key distinction between GANs and DCGANs. In vanilla GANs, the reliance on dense layers makes it difficult to model the spatial relationships inherent in images, leading to poorer visual outputs. DCGANs, on the other hand, excel at image synthesis due to their convolutional architecture, which effectively captures spatial data and patterns. This makes DCGANs particularly well-suited for tasks such as texture generation and high-quality image synthesis.

6.2.3 Input Handling

GANs are versatile in terms of input data. They can accept a wide range of inputs, including matrices, vectors, and higher-dimensional data, making them applicable across different domains. DCGANs, however, are primarily designed for handling image data. Their architecture performs best when dealing with square images, typically with resolutions like 64x64 pixels. This focus on image data enables DCGANs to generate superior visual outputs in applications requiring high-quality imagery.

6.2.4 Pooling Techniques

Pooling methods also vary between the two models. Traditional GANs often utilize techniques such as max-pooling or average-pooling to downsample data within the network. In contrast, DCGANs avoid pooling layers altogether. Instead, they use strided convolutions in the discriminator for downsampling and fractionally-strided convolutions in the generator for upsampling. This approach contributes to the stability and effectiveness of DCGANs in generating detailed images.

6.2.5 Batch Normalization

Another critical difference lies in the use of batch normalization. While traditional GANs may or may not implement batch normalization, DCGANs consistently apply this technique to both the generator and discriminator networks. The use of batch normalization helps stabilize the training process, ensuring that the model converges more reliably and produces better-quality outputs.

6.2.6 Activation Functions

The activation functions used by GANs and DCGANs further distinguish their architectures. Standard GANs commonly employ sigmoid or ReLU activation functions across their layers. DCGANs, however, use LeakyReLU in the discriminator and ReLU in the generator, with the exception of the generator's output layer, where the tanh activation function is applied. This choice of activations enhances the generator's ability to produce more realistic images by better approximating the underlying data distribution.

6.2.7 Training Stability

Training stability is a common challenge for vanilla GANs. They are prone to issues such as training instability, convergence difficulties, and mode collapse, where the generator produces a limited range of outputs. DCGANs address these issues by leveraging their convolutional architecture, which improves stability during training. As a result, DCGANs tend to generate more consistent, realistic, and detailed images, particularly in applications that demand high-quality visual outputs.

6.2.8 Spatial Hierarchies

One of the notable limitations of traditional GANs is their difficulty in capturing spatial hierarchies in image data, which can degrade the quality of the generated outputs. DCGANs, with their convolutional layers, excel at capturing these spatial hierarchies and relationships. This capability enables DCGANs to generate images that are more cohesive and visually accurate, making them a preferred choice for tasks requiring intricate image synthesis.

6.2.9 Image Resolution

Finally, image resolution serves as a major point of distinction. While traditional GANs can generate images at varying resolutions, they often struggle to produce high-resolution images with sufficient detail. DCGANs, on the other hand, are specifically optimized for generating high-resolution images. Their convolutional architecture allows them to capture fine-grained details and textures, resulting in visually compelling outputs, particularly for image-centric applications that require high levels of realism.

In summary, GANs are versatile models used for a wide range of generative tasks, such as style transfer and text-to-image synthesis, showing their adaptability in various fields. In contrast, DCGANs focus primarily on generating high-quality images, making them ideal for tasks like image-to-image translation and picture synthesis within computer vision. Their specialization allows them to produce realistic and detailed images that meet the needs of many visual applications. While GANs are broadly applicable, DCGANs, with their deep convolutional networks, are optimized for realistic image creation, capturing spatial details and improving training stability. capacity to produce realistic, high-quality images, which makes DCGANs very useful for situations requiring intricate image synthesis.

7. The GAGAN - Combining GAs with GANs

7.1 Introduction to GAGAN Hybrid

The integration of Genetic Algorithms (GAs) with Generative Adversarial Networks (GANs) has emerged as a promising approach to enhancing the performance of generative models. GANs consist of two neural networks—the generator and the discriminator—that engage in a competitive game to produce realistic synthetic data. While GANs have been remarkably effective in applications like image generation and style transfer, they often suffer from issues such as mode collapse, instability during training, and suboptimal convergence.

7.1.1 Leveraging Genetic Algorithms in GANs

To address these challenges, the GAGAN hybrid framework leverages the evolutionary optimization capabilities of Genetic Algorithms. In this setup, the weights of the discriminator network are treated as individuals in a genetic population, allowing GAs to explore a broader solution space. By applying selection, crossover, and mutation mechanisms, the GA optimizes the discriminator’s parameters, enhancing its ability to distinguish between real and generated images.

7.1.2 Advantages of the GAGAN Synergy

The synergy between GAs and GANs offers several key benefits. First, GAs' global search capabilities can help GANs navigate complex loss landscapes, leading to improved convergence and more stable training dynamics. Second, the evolutionary diversity introduced by GAs mitigates the risk of mode collapse, a common issue in GAN training, which in turn boosts the quality and diversity of generated outputs.

7.1.3 Exploring the GAGAN Mechanisms

This chapter examines the mechanisms behind the GAGAN hybrid framework, focusing on how genetic operators are applied during GAN training. It also discusses the advantages and limitations of this approach, offering insights into its potential for generating high-quality synthetic data and its implications for the future of generative modeling.

7.2 Mechanism of GAGAN Hybrid

7.2.1 Individual Representation

In the GAGAN hybrid framework, the Genetic Algorithm (GA) treats the weights of the Generative Adversarial Network’s (GAN) discriminator as individuals within its population. These individuals represent potential solutions to the problem of optimizing the discriminator’s performance.

Weights as Individuals: Each individual corresponds to a unique set of weights for the GAN's discriminator. These weights determine how well the discriminator distinguishes real images from generated ones, influencing the overall performance of the GAN.

Vector Representation: The weights of the discriminator are typically flattened into a one-dimensional vector, where each element corresponds to a specific parameter within the model. This transformation allows the genetic operations, such as crossover and mutation, to be applied efficiently across the entire network. For instance, weights from convolutional, batch normalization, and dense layers are concatenated into this vector format.

Dimensionality: The dimensionality of each individual is equal to the total number of trainable parameters in the discriminator, which can be quite high, especially in deep networks. This high-dimensional search space presents a challenge, but it also offers flexibility for the GA to explore a vast array of potential solutions.

Genetic Operations: To explore the weight space, individuals undergo genetic operations like crossover, where parts of two individuals are combined, and mutation, where random changes are introduced. These operations allow the GA to evolve better configurations for the discriminator's weights, gradually improving the GAN's performance.

7.2.2 Fitness Function Definition

The fitness function in GAGAN serves as the key metric for evaluating how effectively an individual's set of weights contributes to the GAN's overall performance. It guides the GA by selecting individuals with better discriminator performance for reproduction and evolution in future generations.

7.2.2.1 Performance Metrics in GAGAN

The fitness function in the GAGAN hybrid framework is typically based on two critical performance metrics: Generator Loss and Discriminator Accuracy. The generator loss measures how effectively the generator produces images that can deceive the discriminator; a lower generator loss indicates that the generated images are becoming increasingly realistic. On the other hand, discriminator accuracy assesses how accurately the discriminator can differentiate between real and generated images, with a higher accuracy reflecting improved performance in distinguishing real from fake. Together, these metrics provide essential insights into the effectiveness of the GAN's training and guide the optimization of the discriminator's weights through the Genetic Algorithm.

Objective: The goal of the fitness function is to strike a balance between minimizing the generator's loss and maximizing the discriminator's accuracy. One way to structure the fitness function could be:

$$\text{Fitness} = \text{Discriminator Accuracy} - \text{Generator Loss}$$

This equation emphasizes both metrics, where lower generator loss and higher discriminator accuracy lead to a higher fitness score.

7.2.2.2 Training Evaluation and Selection Process in GAGAN

The fitness function in the GAGAN hybrid is evaluated during the GAN’s training, typically after a few epochs. At this stage, the individual’s weights are applied to the discriminator, and the GAN undergoes a brief training period to assess the performance metrics that inform the fitness value. Following this evaluation, individuals with the highest fitness values are selected for reproduction in the next generation. This selection process ensures that only the most effective discriminator configurations are carried forward, allowing the population of individuals to evolve towards better performance over successive generations.

The GAGAN hybrid strategy efficiently optimizes the GAN's discriminator by combining these two elements—individual representation and fitness function definition—which may result in enhanced performance in producing realistic images.

7.3 Genetic Operators in GAGAN

In a Genetic Algorithm (GA) combined with Generative Adversarial Networks (GANs), the genetic operators play a crucial role in creating a diverse population of solutions that can effectively optimize the performance of the GAN. This section covers the three primary genetic operators: selection, crossover, and mutation, as they are applied in the GAGAN framework.

7.3.1 Selection Process

The selection process is essential for determining which individuals from the current population will be used to produce offspring for the next generation. In this implementation, tournament selection is employed. In tournament selection, a subset of individuals is randomly chosen from the population. The fitness values of these individuals are evaluated, and the one with the best fitness is selected to produce offspring. This process can be repeated multiple times to select multiple individuals. The advantages of tournament selection include robustness, as it reduces the risk of selecting poor individuals since it only considers the best within a subset, and diversity maintenance, as randomly choosing individuals for each tournament promotes genetic diversity within the population, which is vital for effectively exploring the search space.

7.3.2 Crossover Techniques

Crossover is a genetic operator used to combine the genetic information of two parents to produce one or more offspring. In the GAGAN framework, the weights of the selected individuals are combined using crossover techniques. One such technique is two-point crossover, which involves selecting two points along the weight vectors of the parent individuals. The segments of the vectors between these

points are swapped to create offspring. The general steps involved are to randomly select two crossover points and then create offspring by swapping the segments between the two parents at these points. This technique allows for the mixing of good traits from both parents, potentially leading to better-performing offspring. Crossover promotes exploration in the weight space, allowing for the discovery of potentially superior configurations that combine the strengths of different individuals.

7.3.3 Mutation Strategies

Mutation introduces random changes to the individuals' weight vectors, providing genetic diversity and helping to avoid local optima. In the GAGAN implementation, Gaussian mutation is applied, which involves adjusting the weights of an individual by adding Gaussian noise. The steps are as follows: for each weight in the individual, a random value is generated from a Gaussian distribution (mean = 0, standard deviation = 1), and this random value is added to the original weight. The mutation rate controls the likelihood of each weight being mutated, allowing for a balance between exploration and exploitation. Mutation is crucial in maintaining genetic diversity and allowing the GA to explore new regions of the search space, which can lead to discovering improved weights for the GAN's discriminator.

In summary, selection, crossover, and mutation—three genetic operators in the GAGAN hybrid—cooperate to generate individuals that serve as weights for the discriminator of the GAN. The GA efficiently searches the solution space by utilizing tournament selection, two-point crossover, and Gaussian mutation, which fosters both convergence and diversity toward the best possible solutions for producing realistic images. These operators improve the GAN's performance by iteratively optimizing its design, allowing it to generate outputs of higher quality.

8. GAGAN in Action: Experimental Insights

8.1 Introduction

The practical experiments conducted with the GAGAN framework aimed to explore the optimization capabilities of Genetic Algorithms within Generative Adversarial Networks. The primary objective was to enhance the performance of the GAN's discriminator by fine-tuning its weights through evolutionary strategies.

8.2 Initial Challenges

At the beginning of my experiments with the Genetic Algorithm-Generative Adversarial Network (GAGAN) framework, I faced uncertainty about which parameters to optimize within the GAN architecture. Given the roles of the generator and discriminator, I ultimately decided to focus on optimizing the weights of the discriminator. This choice stemmed from the understanding that the discriminator plays a crucial role in distinguishing real images from those generated by the GAN.

A well-optimized discriminator is essential for providing accurate feedback to the generator, guiding it to produce increasingly realistic images. By enhancing the performance of the discriminator, I aimed to improve the overall efficacy of the GAN, as a strong discriminator not only boosts the generator's learning process but also helps mitigate issues like mode collapse. This strategic focus on optimizing the discriminator’s weights was fundamental to achieving better results with the GAGAN model.

8.3 Experiment Setup

Initially, I utilized Python with the free version of Google Colab, equipped with a T4 GPU, for training the GAGAN model. However, I encountered significant challenges during the training process. After only a few epochs, the program would frequently collapse due to running out of memory or RAM, or it would exceed the available free time for computation. Additionally, the training times were excessively long, often taking several hours, even though I began with a relatively smaller dataset. These issues highlighted the need for a more robust computational setup to effectively train the GAGAN model without interruptions.

8.4 Approach to Optimization

To address these challenges, I engaged in an iterative process of parameter adjustments. I gradually reduced the size of the dataset, alongside modifying other parameters, including the number of epochs, image resolution, latent dimension, and learning rates for both the generator and discriminator. I also experimented with different population sizes (n) and the number of generations (ngen). Despite these adjustments, the training sessions remained time-consuming, often taking between 3 to 7 hours, even with reduced parameters.

After continuous experimentation, I ultimately decided to transition to a paid A100 GPU for training. This change was aimed at facilitating a more efficient training process, allowing me to adapt parameters without prolonging the training duration excessively.

```
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - d_loss: 0.1729 - g_loss: 8.4653
Epoch 44/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - d_loss: 0.1854 - g_loss: 8.2426
Epoch 45/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - d_loss: 0.2157 - g_loss: 8.4754
Epoch 46/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - d_loss: 0.2255 - g_loss: 8.5927
Epoch 47/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - d_loss: 0.3265 - g_loss: 7.8541
Epoch 48/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 61ms/step - d_loss: 0.1691 - g_loss: 8.3898
Epoch 49/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - d_loss: 0.2667 - g_loss: 8.5835
Epoch 50/50
62/62 ━━━━━━━━━━━━━━━━━━━━ 4s 62ms/step - d_loss: 0.1696 - g_loss: 8.2962
```

Figure 28: Training Process of GAGAN

8.5 Final Parameter Configuration

The final configuration that yielded satisfactory results involved a streamlined approach, where I experimented with various parameter settings before settling on the most effective ones. I downsized and utilized a dataset of 2,000 images with a standardized image resolution of 128 x 128 pixels. The population size (n) and the number of generations (ngen) were kept relatively small, which helped to significantly reduce the training time to approximately one hour. This setup allowed me to achieve the desired results without the complications faced with larger datasets

8.6 Integrated Numerical and Visual Analysis of GAGAN and DCGAN Models

In this section, we present a comprehensive analysis that integrates both numerical performance metrics and visual outputs of the GAGAN and DCGAN models. By examining the final losses for multiple iterations of both models, we aim to provide insights into their relative effectiveness in generating high-quality synthetic images.

8.6.1 Numerical Analysis

8.6.1.1 Final Losses

To provide a comprehensive understanding of the performance of both GAGAN and DCGAN models, here are provided some of the final results of the losses recorded during the last epoch of some of the trainings. These losses serve as critical indicators of how well each model has optimized the generator

and discriminator components. Below are the final losses of the generator (g_loss) and discriminator (d_loss) obtained from the experiments:

- **GAGAN 1:** Epoch 50/50 - d_loss: 0.1696, g_loss: 8.2962
- **GAGAN 2:** Epoch 50/50 - d_loss: 0.1882, g_loss: 29.2821
- **GAGAN 3:** Epoch 50/50 - d_loss: 0.0676, g_loss: 35.2944
- **GAGAN 4:** Epoch 50/50 - d_loss: 0.1383, g_loss: 5.1635
- **DCGAN 1:** Epoch 50/50 - d_loss: 0.4647, g_loss: 3.0347
- **DCGAN 2:** Epoch 50/50 - d_loss: 0.2854, g_loss: 2.5461
- **DCGAN 3:** Epoch 50/50 - d_loss: 0.3585, g_loss: 2.1455
- **DCGAN 4:** Epoch 50/50 - d_loss: 1455445.5000, g_loss: 5114314.0000

8.6.1.2 Observations

From the numerical results, several key observations can be made:

Discriminator Loss Comparison:

- The GAGAN models exhibit lower discriminator loss values compared to the DCGAN models in most cases, indicating that the GAGAN's discriminator was more effective at distinguishing real from generated images. For instance, GAGAN 1 and GAGAN 4 achieved d_loss values of 0.1696 and 0.1383, respectively, compared to the DCGAN 1 value of 0.4647. This suggests that GAGAN is better at optimizing the discriminator's performance.

Generator Loss Comparison:

- The generator loss for GAGAN models varies significantly, with GAGAN 3 showing a notably high value of 35.2944, which might indicate that it struggled to produce images that sufficiently fooled the discriminator. However, GAGAN 4 achieved a g_loss of just 5.1635, suggesting it was able to generate more convincing images.
- In contrast, DCGAN models showed consistently lower generator loss values, especially DCGAN 1 and DCGAN 2, with g_loss values of 3.0347 and 2.5461, respectively. This indicates that while the generator was somewhat effective, it did not match the performance of GAGAN in generating high-quality images.

Issues in DCGAN Performance:

- The extremely high losses recorded for DCGAN 4 (d_loss: 1455445.5000, g_loss: 5114314.0000) suggest a potential issue during training, such as failure in the

- optimization process or mode collapse. This raises questions about the stability of the DCGAN training process, particularly in larger datasets or complex image distributions.

8.6.2 Visual Comparison

Below are the images generated by both GAGAN and DCGAN models, illustrating the differences in output quality despite the comparable training parameters.

8.6.2.1 Images Generated with GAGAN

Figure 29: Generated Images with GAGAN

8.6.2.2 Images Generated with DCGAN

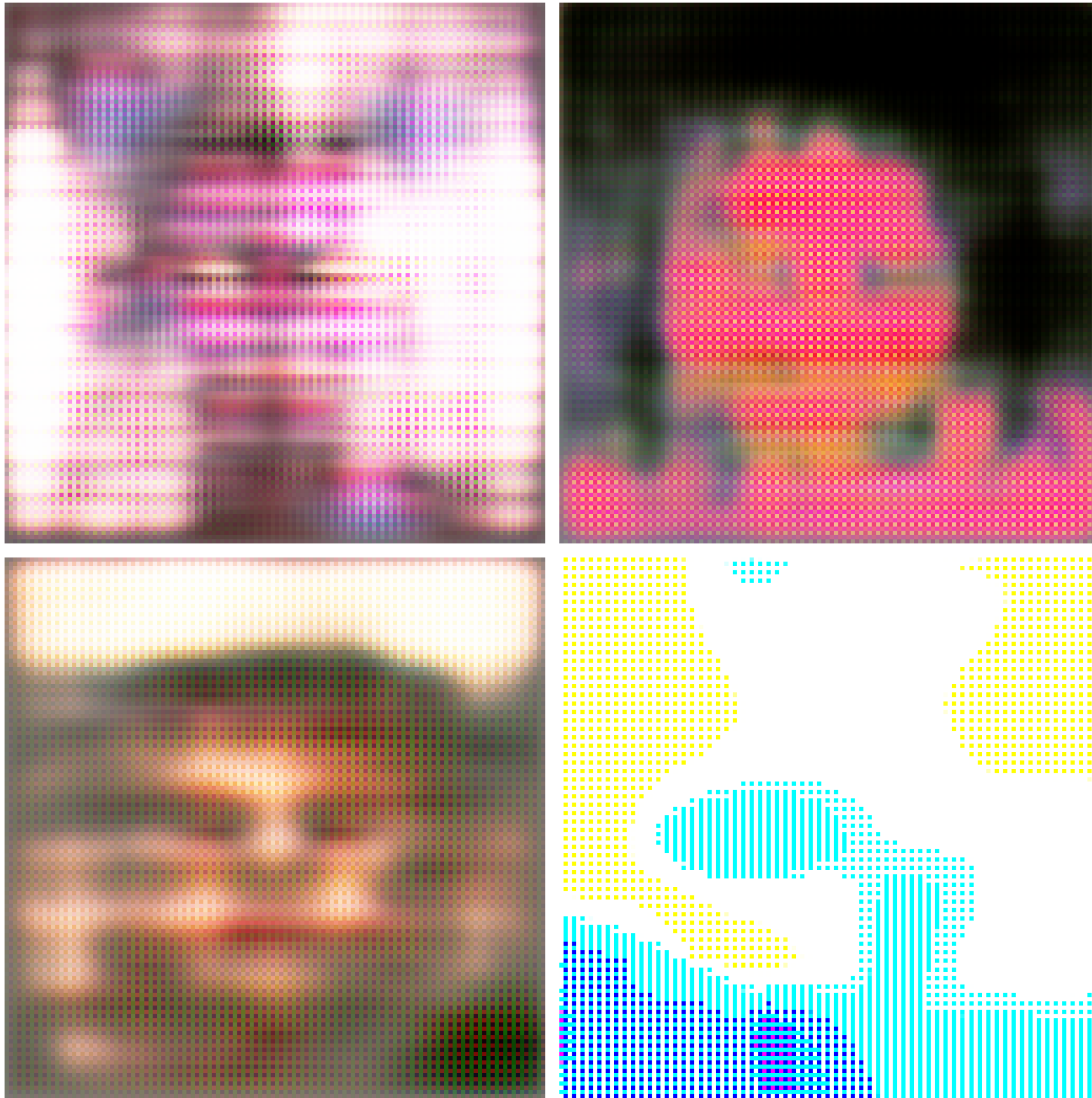


Figure 30: Generated Images with DCGAN

8.6.2.3 Visual Outputs

Alongside the numerical analysis, we also examined the visual outputs generated by both models. The images produced by GAGAN demonstrated greater detail and variation compared to those generated by DCGAN, highlighting the effectiveness of the genetic optimization process in enhancing the generator's capability.

- **GAGAN Images:** The images from GAGAN were not only more realistic but also displayed a wider range of styles and features, suggesting that the evolutionary approach effectively diversified the generator's outputs.
- **DCGAN Images:** The images produced by DCGAN were less varied and often exhibited artifacts, which are indicative of lower-quality generation. The difference in quality becomes apparent when directly comparing similar classes of images generated by both models.

The combination of numerical and visual analyses clearly indicates that GAGAN outperforms DCGAN in generating higher-quality images while maintaining effective discriminator performance. The evolution of the discriminator weights through the genetic algorithm appears to provide significant advantages, contributing to both improved image realism and diversity.

8.7 Results and Observations

The results from the experiments demonstrated that the strategic adjustments made to the parameters, particularly the focus on optimizing the discriminator, significantly enhanced the performance of the GAGAN model. Utilizing the A100 GPU not only made the training process more efficient but also allowed for faster iterations, facilitating a deeper exploration of the weight space. This optimization led to noticeable improvements in the quality of generated images and overall model stability.

8.8 Conclusion

In conclusion, the practical experiments conducted with GAGAN provided profound insights into the interplay between Genetic Algorithms and Generative Adversarial Networks. Despite facing initial hurdles related to parameter selection and computational constraints, the iterative refinement process ultimately yielded successful training outcomes. This journey highlights the critical role of flexibility and informed decision-making in the field of generative modeling, showcasing how a thoughtful approach to optimization can lead to significant advancements in model performance. Through this work, we have not only enhanced the GAGAN framework but also contributed to a deeper understanding of the dynamics within generative modeling, paving the way for future explorations and improvements.

9. Challenges and Future Directions

The integration of Genetic Algorithms with GANs posed several challenges, primarily related to computational limitations, parameter optimization, and training efficiency. While progress was made, further improvements and refinements are essential. This section outlines the key obstacles encountered and explores potential future directions for advancing the GAGAN framework.

9.1 Challenges

9.1.1 Computational Constraints and Resource Limitations

One of the key challenges encountered during the GAGAN experiments was the limitation of computational resources. The initial implementation relied on the free version of Google Colab, using a T4 GPU, which presented significant obstacles. Frequent program collapses due to memory exhaustion, RAM shortages, or exceeding the free usage time were common occurrences. While transitioning to a paid A100 GPU alleviated these issues, the complexity and time-intensive nature of GAGAN training remained a hurdle, especially in the context of optimizing the discriminator with Genetic Algorithms. This revealed the critical need for more efficient computational strategies to make GAGAN training more accessible and less resource-dependent.

9.1.2 Balancing Dataset Size and Training Time

Another challenge was balancing the size of the dataset with the duration of the training process. Although smaller datasets helped reduce training time, they often resulted in lower-quality generated images. On the other hand, larger datasets increased training times significantly, sometimes taking several hours to complete, even with optimized parameters. This trade-off between speed and quality remains an ongoing challenge in GAGAN training. Future improvements could focus on more efficient training techniques that maintain image quality while reducing the time investment required.

9.1.3 Parameter Selection and Model Tuning

The process of selecting and fine-tuning parameters for GAGAN was another challenge. The initial uncertainty about which parameters should be optimized within the GAN framework led to some trial-and-error experimentation. Ultimately, focusing on the discriminator’s weights yielded the best results, but this required extensive testing of different configurations. The need for continuous parameter tuning—adjusting the population size, number of generations, learning rates, and other settings—highlighted the complexity of optimizing GANs with evolutionary strategies. Future work could explore automated tuning mechanisms or meta-learning techniques to streamline this process.

9.2 Future Directions

9.2.1 Advanced Genetic Algorithms

To address the challenges of slow convergence and model instability, future research could focus on developing more sophisticated genetic algorithms for GAGAN. For instance, incorporating adaptive mutation rates that dynamically change based on the model's performance could help balance exploration and exploitation more effectively. Furthermore, introducing more complex crossover techniques could enable the discovery of more optimal configurations within the discriminator's weight space, resulting in higher-quality generated images and more stable training.

9.2.2 Integrating Perceptual Metrics for Fitness Evaluation

Another potential area for improvement is the fitness function used to guide the genetic algorithm. In the current implementation, the fitness function is based on loss values, which do not always capture the subjective quality of generated images. Future research could incorporate perceptual metrics, such as those based on human visual perception or deep learning-based image quality assessments, into the fitness evaluation. This would allow for a more comprehensive assessment of image quality and guide the evolution of models toward producing more visually appealing results.

9.2.3 Expanding Genetic Optimization to the Generator

While the current research focused on optimizing the discriminator's weights, future work could expand the scope to include the generator's weights as well. Optimizing both networks in tandem could result in more balanced training dynamics and improved overall performance. Developing strategies for evolving generator weights, in combination with discriminator optimization, could lead to a more comprehensive approach to improving generative models through evolutionary algorithms.

Conclusions

This paper explored the integration of Genetic Algorithms into Generative Adversarial Networks (GAGAN), focusing specifically on optimizing the discriminator's weights. The entire framework was developed and implemented in Python, which provided flexibility for experimenting with various parameters and algorithms. After extensive testing, it was determined that optimizing the discriminator, given its essential role in distinguishing real from generated images, was key to improving the model's overall performance.

The practical experiments presented several challenges, notably computational constraints, particularly when using the free version of Google Colab, and difficulties related to dataset size, long training times, and parameter tuning. By shifting to the more powerful A100 GPU, significant improvements were achieved in training efficiency, allowing for a more balanced configuration that optimized both time and performance.

Through multiple iterations, different combinations of population size, the number of generations, and other parameters were tested, resulting in a final GAGAN configuration that produced higher-quality images compared to traditional GANs, though with the trade-off of longer training times. Despite the complexity and time-intensive nature of the process, the model showed the potential for significantly enhancing generative modeling through the use of evolutionary optimization.

These findings lay a strong foundation for future work. Potential areas for improvement include optimizing both the generator and discriminator simultaneously, introducing more sophisticated genetic operators, and incorporating perceptual metrics for a more accurate fitness evaluation. With these enhancements, further advancements in the efficiency and performance of GAGAN models are expected to emerge, pushing the boundaries of generative modeling even further.

Bibliography- References- Papers- Online Sources

[1] https://www.researchgate.net/publication/262818282_Computational_Intelligence_Based_Technique_in_Optimal_Overcurrent_Relay_Coordination_A_Review

[2] https://cis.ieee.org/about/what-is-ci

[3] https://www.mdpi.com/2218-6581/12/4/106

[4] https://www.researchgate.net/figure/Evolutionary-computation-family-of-algorithms-The-common-evolutionary-computation_fig1_332798355

[5] https://en.wikipedia.org/wiki/Evolutionary_algorithm

[6] https://link.springer.com/referenceworkentry/10.1007/978-3-540-92910-9_26

[7] https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac

[8] https://www.sciencedirect.com/topics/computer-science/simple-genetic-algorithm

[9] https://academic.oup.com/mind/article/LIX/236/433/986238

[10] https://en.wikipedia.org/wiki/Genetic_algorithm

[11] https://www.geeksforgeeks.org/genetic-algorithms/

[12] https://www.javatpoint.com/genetic-algorithm-in-machine-learning

[13] https://link.springer.com/article/10.1007/s11042-020-10139-6#Sec3

[14] https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[15] https://www.slideshare.net/slideshow/explanation-and-example-of-genetic-algorithm/267426880#2

[16] https://medium.com/@byanalytixlabs/a-complete-guide-to-genetic-algorithm-advantages-limitations-more-738e87427dbb

[17] https://www.curriculumonline.ie/getmedia/96dfcf0a-8373-48d3-9662-714ab8c8a428/NCCA-The-Evolution-of-Computers-in-Society-LC-SC-the-turing-machine.pdf

[18] https://arxiv.org/pdf/1904.05061

[19] https://www.britannica.com/science/history-of-artificial-intelligence

[20] https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec

[21] https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/

[22] https://en.wikipedia.org/wiki/Neural_network

[23] https://medium.com/@sedatparlak1953/foundation-of-neural-networks-a68925aa7e2

[24] https://h2o.ai/wiki/weights-and-biases

[25] https://www.researchgate.net/figure/The-basic-structure-of-the-Artificial-Neural-Network-ANN-model-input-hidden-and_fig1_331281246

[26] https://en.wikipedia.org/wiki/Activation_function

[27] https://www.v7labs.com/blog/neural-networks-activation-functions

[28] https://www.datacamp.com/tutorial/loss-function-in-machine-learning

[29] https://www.geeksforgeeks.org/regularization-in-machine-learning/

[30] https://www.deeplearningbook.org/

[31] https://svitla.com/blog/modern-methods-of-neural-network-training/

[32] https://www.geeksforgeeks.org/neural-networks-a-beginners-guide/#learning-of-a-neural-network

[33] https://www.mygreatlearning.com/blog/types-of-neural-networks/

[34] https://stackademic.com/blog/the-difference-between-back-propagation-and-forward-propagation-in-deep-learning-2b2248e6d00c

[35] https://medium.com/@nerdjock/deep-learning-course-lesson-5-forward-and-backward-propagation-ec8e4e6a8b92

[36] https://www.geeksforgeeks.org/introduction-convolution-neural-network/

[37] https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns

[38] https://en.wikipedia.org/wiki/Convolutional_neural_network

[39] full article: https://www.nature.com/articles/s41598-024-51258-6#Fig1

[40] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[41] https://www.datacamp.com/tutorial/cnn-tensorflow-python

[42] https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

[43] https://machine-learning.paperspace.com/wiki/overfitting-vs-underfitting

[44] https://en.wikipedia.org/wiki/Generative_adversarial_network

[45] https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

[46] https://www.mdpi.com/1424-8220/24/2/641

[47] https://www.techtarget.com/searchenterpriseai/definition/generative-adversarial-network-GAN

[48] https://www.simplilearn.com/tutorials/deep-learning-tutorial/generative-adversarial-networks-gans#how_do_gans_work

[49] https://www.researchgate.net/publication/348837975

[50] https://docs.chainer.org/en/stable/examples/dcgan.html

[51] https://www.leewayhertz.com/generative-adversarial-networks/#Future-of-Generative-Adversarial-Networks

[52] https://medium.com/aimonks/an-introduction-to-generative-adversarial-networks-gans-454d127640c1

[53] https://link.springer.com/article/10.1007/s00521-021-05982-z

[54] https://towardsdatascience.com/dcgans-deep-convolutional-generative-adversarial-networks-c7f392c2c8f8

[55] https://arxiv.org/pdf/1511.06434, also figure 26

[56] https://arxiv.org/pdf/1406.2661, also figure 27

[57] https://www.geeksforgeeks.org/difference-between-gan-vs-dcgan/

[58] https://bamos.github.io/2016/08/09/deep-completion/